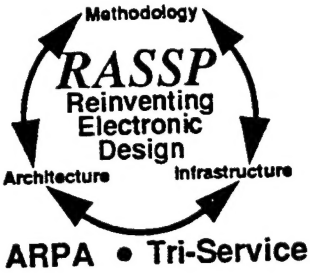
WL-TR-96-1022

GRAPH TRANSLATION TOOL (GRTT) USER'S MANUAL



ARPA • Tri-Service

CHRISTOPHER ROBBINS

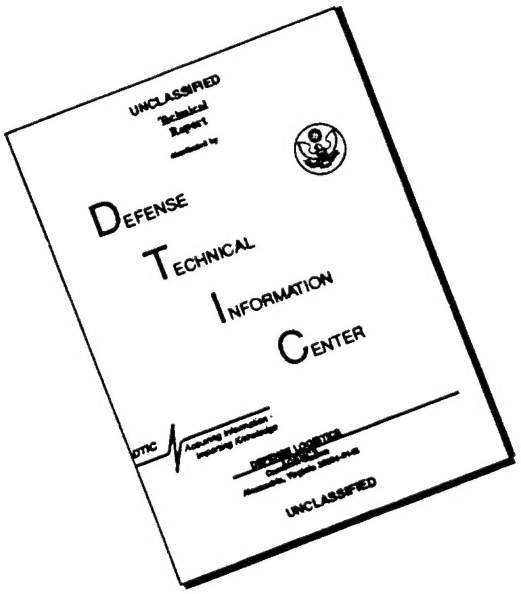
APRIL 1996

FINAL REPORT APRIL 1996

Approved for public release; distribution unlimited

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7623

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

GARY FECHER, Project Engineer
Computer Aided Engineering Tech Sec
System Technology Branch

DAVID A. ZANN, Chief
System Technology Branch
System Concepts & Simulation Div

STANLEY E. WAGNER, Chief
System Concepts & Simulation Div
Avionics Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/AAST-1, WPAFB, OH 45433-7623 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	APRIL 1996	FINAL APRIL 1996

4. TITLE AND SUBTITLE

GRAPH TRANSLATION TOOL (GrTT) USER'S MANUAL

5. FUNDING NUMBERS

C F33615-94-C-1559
PE 63739E
PR A268
TA 02
WU 19

6. AUTHOR(S)

CHRISTOPHER ROBBINS

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

MANAGEMENT COMMUNICATIONS AND CONTROL, INC.
2000 NORTH 14th STREET, SUITE 220
ARLINGTON, VIRGINIA 22201

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7623

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

WL-TR-96-1022

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document describes the use of the Graph Translation Tool (GrTT), a software program for translating a signal processing graph, expressed in the Processing Graph Method (PGM), into Ada source code that implements the signal processing embodied by the graph. PGM is the Navy's standard for signal processing specification methodology, developed to provide a programming environment for signal processing graphs for the AN/UYS-2

14. SUBJECT TERMS	15. NUMBER OF PAGES
Graph Translation Tool, Processing Graph Method, AN/UYS-2	55
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

Graph Translation Tool (GRTT) Final Report

Contents

GrTT - Graph Translation Tool
Final Report

Introduction

The Graph Translation Tool (GrTT) is a member of Management Communications and Control, Inc.'s (MCCI) autocoding toolset. Autocoding provides the hardware/software codesign process the means to rapidly realize implementations of the codesign software architectures. MCCI's autocoding tools automate translation of software architecture specifications to designs, their behavior models, and their implementations. The toolset supports behavior model generation, functional and performance hardware/software cosimulation, unit testing, and complete application load image specification. The tools support an open application programmer's interface to the codesign process. Autocoding tools support a major objective of the codesign process which is to provide a seamless translation of applications from math tool level functional algorithm specifications to target architecture load images. Autocoding technology is directed at reducing the labor content of software design and coding, enabling rapid development of the software elements of application specific signal processing systems.

GrTT provides the capability to generate behavior models of the hardware/software codesign partitioning. GrTT translates both software and hardware partition data flow graph specifications to behavior models that may be used for requirements validation and test vector generation in support of unit testing. This technical report on the GrTT development project describes both the tool and its role in the autocoding process within the Lockheed Martin ATL•Camden RASSP Enterprise System. The report includes:

- a description of the autocoding process,
- a description of the role GrTT plays in the autocoding process,
- the GrTT concept of operations including the architecture of the tool and its functions, and an example of its use
- the user's manual

Examples of Ada behavior models generated by GrTT are included as an attachment to this report.

Autocoding Overview

The autocoding toolset provides automated assistance for realizing top level software designs from architecture and signal processing data flow graph specifications, and it fully automates detailed software design and coding. Figure 1 illustrates a notional HW/SW codesign process. The process provides for (1) system signal processing and control functional definition; (2) architecture definition; (3) automated application software detailed design and coding; and (4) software integration and test. These steps include domain engineering activity, where processing function, control, and architecture

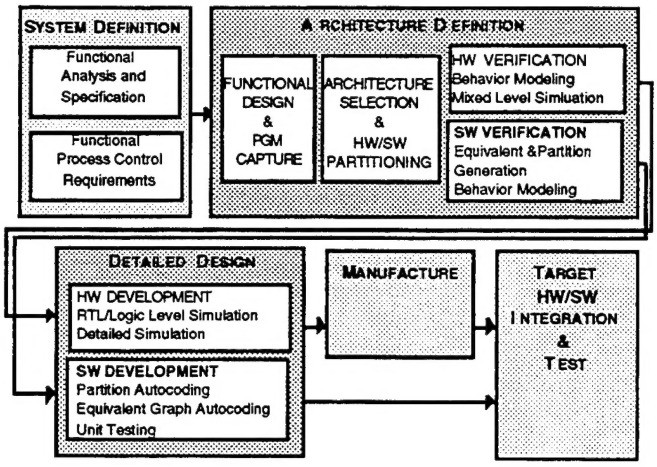


Figure 1. - HW/SW Codesign Process

specifications are developed, and architecture implementation activity at the modeling, verification, detailed design, and coding levels. Autocoding tools are focused on the traditionally labor intense behavior modeling, software verification, software detailed design, and software coding activities. Architecture specifications are automatically translated into executable partition and equivalent application specifications in response to the domain engineer's partition and control parameter specifications. Behavior models are created and functional requirements captured by architecture partitions are validated. Codesign performance is simulated. Performance estimation feedback is nearly instantaneous and performance validation via architecture simulation rapidly follows. Verified software partition specifications are automatically translated to source code for the partition executables. Verified software equivalent application specifications are automatically translated to source code and data structures by which the run-time system controls the execution of the equivalent application specifications.

The autocode toolset provides an open application programmer's interface (API). Figure 2 illustrates the elements of this open API. The algorithm functionality is captured in Programming Graph Method (PGM) data flow graphs. PGM is a Navy developed standard. PGM graphs exist in iconic and notational form. PGM supports specification of full system data flow and data flow control. Processing functions are specified by graph nodes. Queues specify data flow between processing nodes. Formal queues and variables may be externally controlled. A command message interface to the graph provides the control interface. Functional process control requirements of the system definition are implemented as sequences of command procedures directed by command messages. Signal processing is specified by domain primitives which are target independent functional signal processing primitive specifications. The autocoding toolset implements domain primitives on each type of computational element supported in the model year architecture. Software architectures specified using the open API may be ported among all model year architectures without change to architecture specification or external controls.

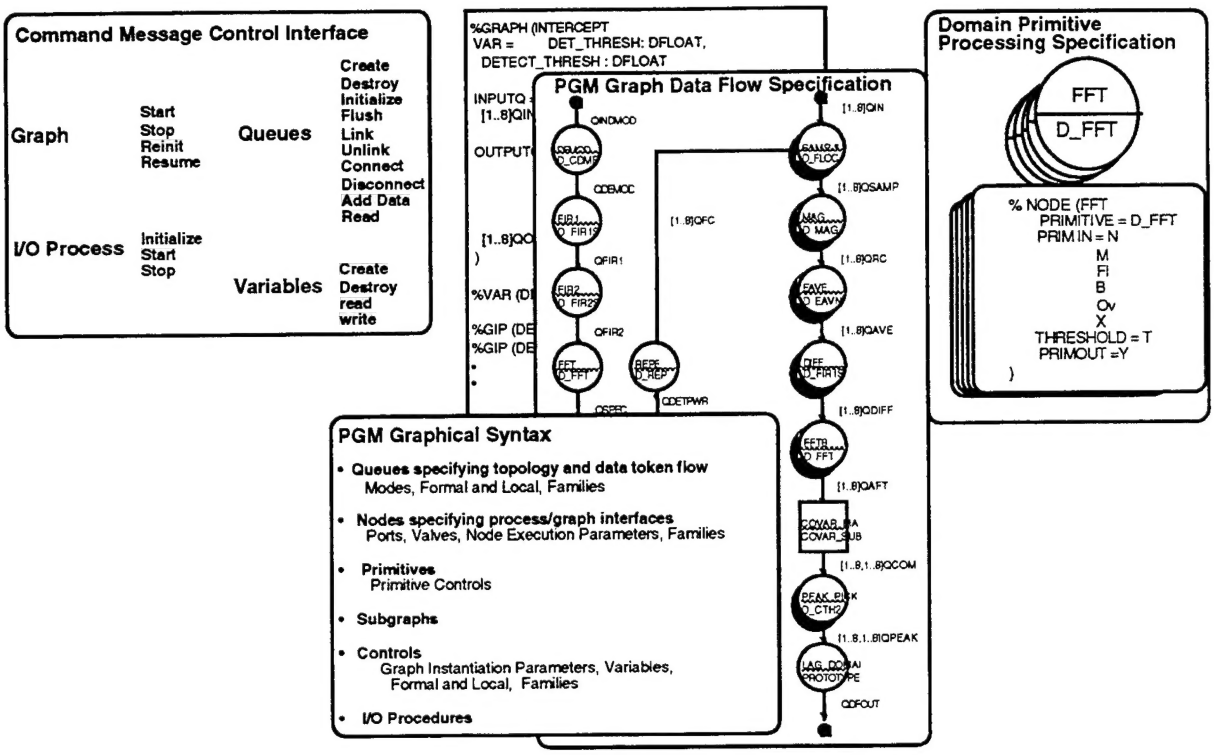


Figure 2. - Open Application Programmer's Interface

Correct by construction software development disciplines are incorporated into the codesign process. Applications are specified and autocoded using the elements of an extensively tested domain primitive software library. These elements are assembled into executable form using mature PGM data flow rules. Validation and/or unit testing activities are incorporated at each stage of codesign to ensure compliance with functional and performance requirements. The autocoding toolset supports the goal of uneventful integration and testing. GrTT is the autocoding tool that most directly supports unit validation. Behavior models are generated for each application software or hardware application partition. Execution of the behavior models on test vectors used in the functional design and simulation of the elements of architecture design validate all of the requirements captured in the top level software design and hardware partition specifications. Test vectors from validated partitions are then reusable to ensure valid implementation of the software architecture specification at every level of the autocoding process.

The autocoding toolset automates system realization of software specifications. It provides tools for validation of codesign architecture specifications and verification of automatically generated implementations of their software elements. It provides automation support to design realization of the application software at architecture verification and detailed design levels of the codesign process. This enables rapid realization of software/architecture virtual prototypes, unit testable application modules, and specifications for compilable system load images on supported targets. Autocoding tools provide the

application domain engineer with the power to expand the number of application/architecture specification variations that can be evaluated within real project budgets. Superior systems and reduced development and/or recurring costs must follow.

Autocode Tools Automate Software Design Realization and Software Architecture Verification. Equivalent and partition graph generation tools automate the generation of top level software designs from hardware/software architecture specifications. As illustrated in Figure 3 hardware/software architectures are specified to the software verification process as PGM application data flow graphs with a candidate architecture file and corresponding partition lists. Partition lists map application graph partitions to architecture processors. A top level software design is generated from these inputs. The top level design consists of (1) an equivalent application data flow graph which specifies the executable image of the application, and (2) a stand-alone partition graph for each equivalent node specifying executable processing. Both outputs may be used to verify requirements capture and performance of the top level software design. Behavior models may be created for each partition using the GrTT tool. The executable procedures from the behavior models may be encapsulated as in the equivalent application graph nodes for functional modeling of the executable image specification. When

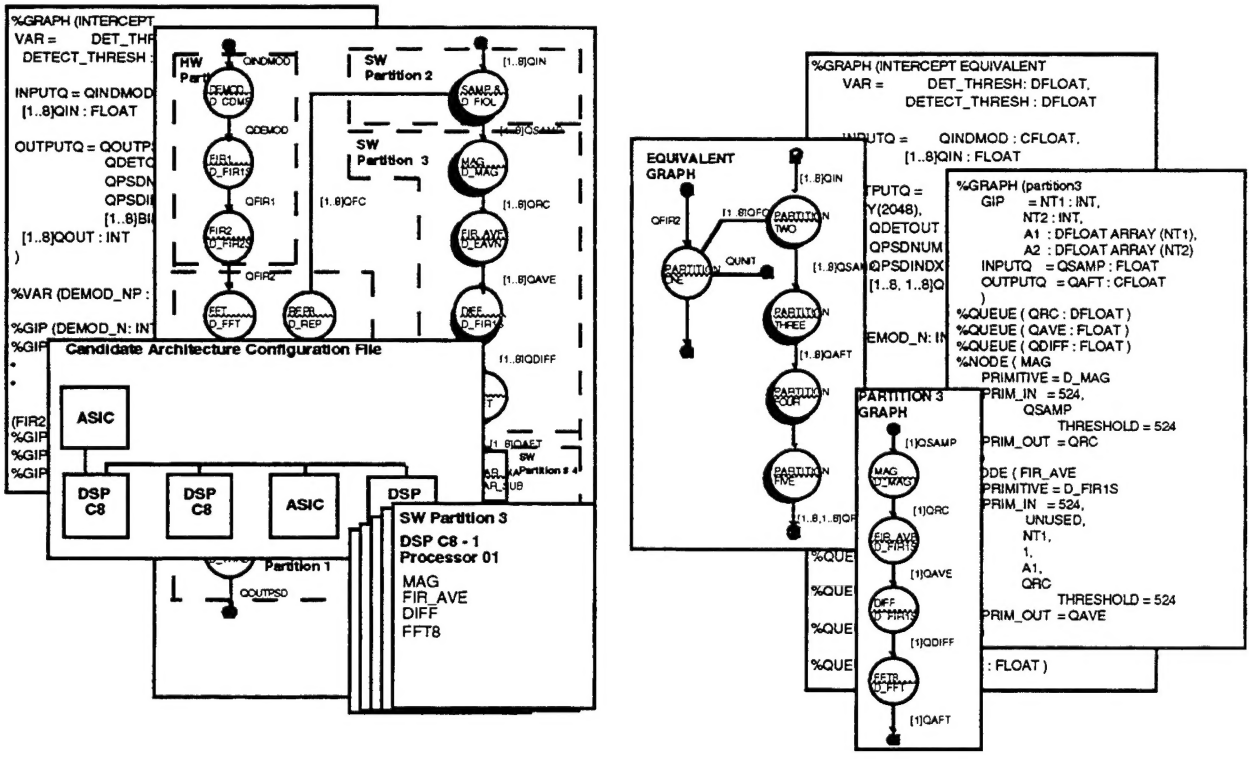


Figure 3. - Equivalent and Partition Graph Generation from Architecture Specifications

accepted, the partitions specified may become inputs to the detailed design level autocode tools that generate application and executable DSP program detailed designs.

Using the equivalent application generator tool, the software designer generates equivalent and partition graphs from the software architecture graph and input software partitions. Partitions may be identified by the partition lists received from the architecture specification tool. Additionally, partitions could be externally generated by some other method or otherwise specified by the autocoding user. Performance estimates are generated consisting of execution time estimates for each partition and equivalent graph execution time and data transfer requirements. Application specifications that will not translate to efficient run-time images may be quickly rejected. Acceptable application designs may be passed to the architecture simulation tools for detailed performance verification. The software designer may iterate partition subdivision and corresponding data flow control parameterization to obtain a software design for a given architecture specification that best meets requirements for minimum resource utilization, load balancing, latency, and memory constraints.

Behavior models are created for each partition graph generated in the top level design process with the GrTT tool. Behavior models provide the functional link between functional behavior of the algorithm as validated on a functional simulator and the behavior of the CSUs of the detailed design. GrTT generated behavior models are the executable requirements specifications for the autocoded implementation of the design's partitions on the target DSP processors.

Performance simulation of the top level software design will be supported by the architecture simulation tool. Partition timing estimates of each equivalent node will be passed to the architecture simulator for hardware/software performance simulation. At the completion of architecture/software design, top level specifications exist from which executable code targeted for the embedded or candidate high performance architecture may be automatically generated. The software verification level autocoding tools automate the generation of equivalent and partition graph specifications. The error prone, laborious, hand coding of top level specifications will be eliminated. Codesigns may be realized at the rate at which designers can make design decisions and evaluate their consequences permitting more design options to be examined..

Autocode Tools to Automate Detailed Design and Coding. Detailed design level autocode tools include the Multi Processor Interface Description (MPID) Generator and the Application Generator. These tools generate compilable images of partition and equivalent graph elements of the top level design. The role these tools play in detailed design is illustrated in Figures 4 and 5.

MPIDs are compilable programs that implement the processing specified by the partition graphs. Both transient, or start up, and cyclic behavior of the partition graphs is preserved in the translation to compilable form as is the partition

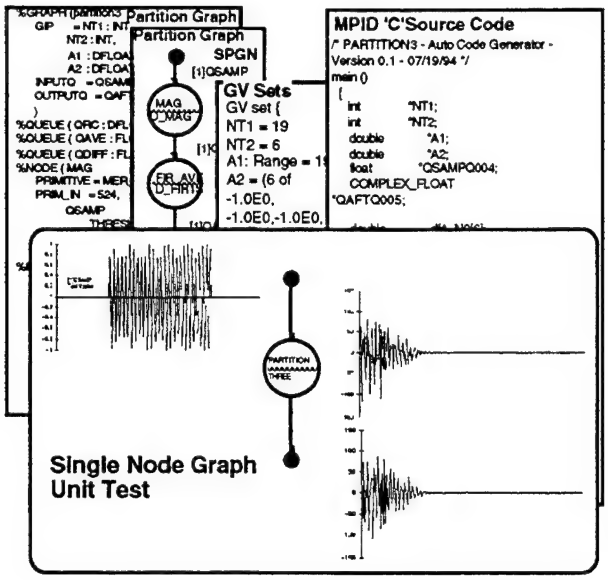


Figure 4. - MPID Generator

graph's response to all enumerated values of controls. At its ports, the execution behavior of the compiled MPID will be identical to the functional behavior of its partition graph specification. MPID Generator will generate 'C' source code implementing the partition's processing specifications utilizing calls to the target processor's math library. A memory map converting all partition internal queues and variables to static buffers is generated. MPID Generator is supported by a domain primitive database which provides constraint, error condition, target specific state machine behavior, and target performance data for each domain primitive. MPID source code will be made as efficient as possible by maximizing in-place execution of target math library calls and minimizing non-library call code to that needed to interface to the equivalent application graph and to respond to external controls.

In addition to source code for the executables, MPIDGen produces detailed performance estimates and single node equivalent graphs specifying the MPID as the primitive. The detailed performance estimates are used to validate software verification performance estimates. The single node graph supports unit testing. Unit test applications are generated using the single node equivalent graph. Test vectors generated by the GrTT behavior model for the partition are processed to validate partition translations. Side-by-side execution of the GrTT behavior model and single node test graph is possible supporting a thorough validation of MPIDs under representative data and external controls. Because of PGM's determinism, validation of each partition implies validation of the full application.

The Application Generator translates the equivalent application graph with its set of MPID source files into data structures that are used by the run-time system to create an executable image of the application as distributed tasks on the target processors. The run-time data structures incorporate the MPIDs as executable elements of the tasks and provide other memory management and

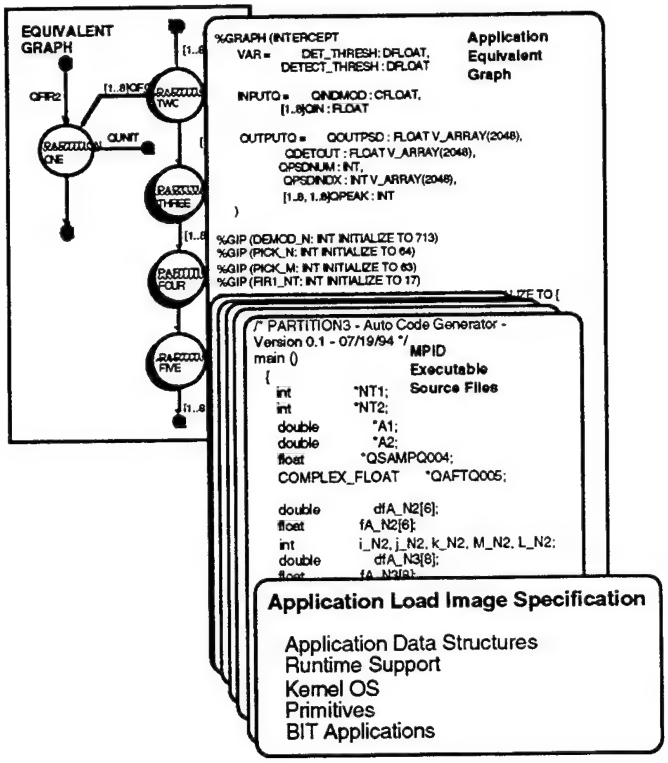


Figure 5. - Application Generator

execution control information needed to realize a run-time image of the equivalent application graph.

Reusable run-time support is provided as part of the application. Figure 6 illustrates the organization of the run-time system into user-supplied signal processing and BIT applications, reusable application and load managers, and operating system kernels. Application data structures provide the interface to the graph manager. The graph manager also controls the command message port. In response to command messages from the Command Program Graphical User Interface (CP GUI), the graph manager instantiates applications as tasks, connects them to data sources and sinks, initiates their processing, and applies all external controls to modify their processing. BIT applications will be handled similarly. Interfaces to operating system software are low level, simplifying the port of the run-time support to new model year computational elements.

A make file is generated by the Application Generator that specifies the load image at the source and data structure level. This make file specifies the source code and executable files containing application data structures, MPIDs, run-time support, and BIT applications. Libraries for kernel OS and target primitives are also specified.

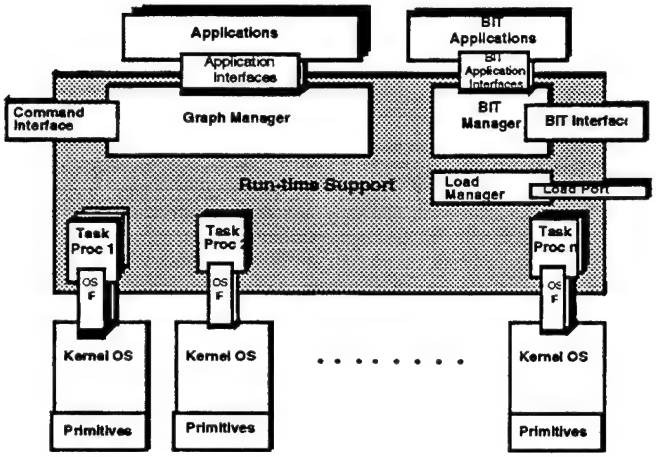


Figure 6. - Run-time Support

Autocoding Tools Reduce Application Engineering Costs. The MCCI autocoding tools will reduce the amount of labor required to generate top level software designs from architecture specifications and detailed designs implementing them. Manual coding will be eliminated altogether. Automated generation of software designs from architecture specifications will allow meaningful evaluation of many alternate designs at a fraction of the time currently required to design signal processing systems. Automated detailed design and code generation will provide testable unit and system implementations of software designs virtually instantly compared to hand coding approaches. Systems representing a thorough design space trade off of alternative application specifications, hardware and software architecture specifications, and lower level partitioning and parameter trades will be produced rapidly. Reusable run-time support avoids an expensive development effort. Reuse of the run-time system provided as part of the reuse library and model year architecture will eliminate the user's need for expensive run-time scheduling and control support development for their software designs. The open architecture supports legacy code capture, model year application porting, and application reuse. As enterprises gain legacy, application reuse opportunity will increase.

Role of Graph Translation Tool (GrTT) in Autocoding

GrTT is an autocoding tool that will translate Processing Graph Method (PGM) graphs to Ada behavior models. GrTT may be used to create behavior models of either hardware or software partitions of PGM data flow graphical application specifications. The functional behavior of the model will be identical to the graph partition represented. Identical outputs will be produced by either model execution or data flow execution of the processing graph on a common input data set. A dynamic view of model execution is supported thus providing visibility of the modeled graph's execution behavior.

Implementation of the RASSP HW/SW codesign process by the Lockheed Martin ATL•Camden RASSP Enterprise System utilizes PGM for data flow specification of the application. Processing within the PGM graph's nodes is specified by domain primitives. Domain primitives are target independent signal processing and data flow control function specifications. Their use in the PGM application specification provides an open application programmer's interface (API) to the team's tools implementing the architecture selection and design processes. Domain primitive graphs are partitioned by the architecture tools into hardware and software allocations. The allocations are further partitioned to become either hardware component partitions or software partitions. Software design tools will generate stand-alone PGM graphs for each partition. GrTT may be used to generate partition behavior models for each hardware or software partition.

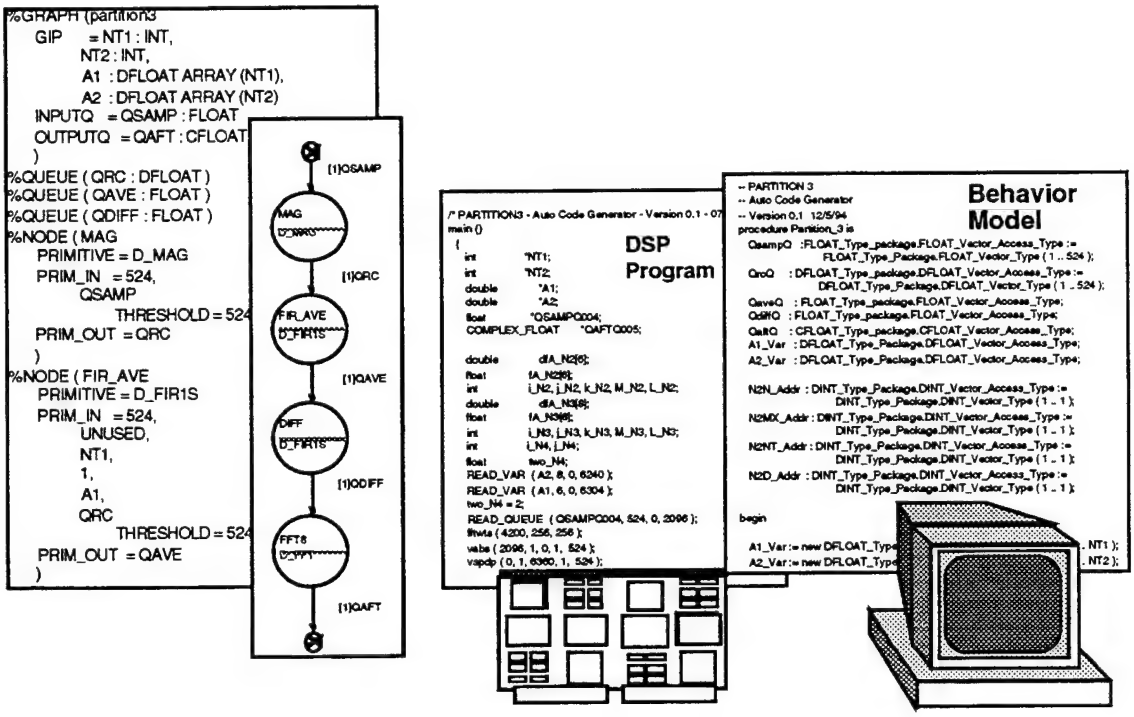


Figure 7. - GrTT Behavior Modeling of PGM Graphs.
GrTT produces an Ada behavior model of a DSP program translated from a PGM graphical specification.

Figure 7 illustrates the partition modeling concept. An application partition graph is shown on the left in both iconic and notational form. Each node has its unique name above the line and specifies the domain primitive implementing the node below the line. Queues represent FIFO buffering of the data between the nodes. Node execution parameters associated with the node ports that are linked to queues specify a thresholding criteria for node execution, data amounts to be read, and data amounts to be consumed from the queues upon node execution. Data amounts produced onto output queues per node execution are functions of the domain primitive controls, read amounts, and

data modes. Node execution parameters, process controls, and parameters may be made run-time variables and provide the capability to externally modify graph execution. Data flow execution, execution of nodes when thresholding criteria are met, guarantees determinism or causal behavior of the graph. GrTT accepts application partition graphs in their notational form plus sets of enumerated values of graph variables, and it produces an Ada procedure that is the behavioral equivalent of the input graph. Graph variables that cause the input graph to alter data flow, node firing rates, or primitive processing will cause identical behavior in the behavior model that GrTT produces.

GrTT consists of three major objects and is supported by the domain primitive database. The translation process these objects implement is illustrated in Figure 8. The domain primitive database provides support for both target independent and target dependent implementations of partition graphs specified with domain primitives. The SPGN parser accepts a partition graph SPGN file and enumerated graph variable (GV) sets. The parser creates a validated graph object, a data structure representing the input graph. Error checking detects any invalid SPGN. All values of variables affecting primitive execution are validated against constraints and requirements of the domain primitives. The graph object represents a flattened graph in which all subgraphs and family constructs have been expanded. GrTT's graph analysis object creates a state machine behavior specification from the graph object and behavior data provided by the domain primitive database. Any behavior error conditions are determined at this point. An example of such an error might be a graph with a periodic execution sequence that would be too long to code or would require too large a memory map. This long periodic execution sequence is normally caused by an ill advised combination of node execution parameters. GrTT's autocoder object generates an Ada procedure implementing the state machine specification for all GV value sets. This Ada procedure becomes the primitive for an equivalent node replacing the partition in the original domain primitive application. A single equivalent node graph containing the procedure as its primitive is also generated by the autocoder object. This single equivalent node graph is useful for unit testing.

The behavior models generated by GrTT may be used to fulfill several important HW/SW codesign functions. GrTT software partition behavior models may be used to validate target specific autocoded executables. The single node graph with a GrTT behavior model embedded as its primitive may be used to validate the partition translation and generate test vectors for other target specific translations of the team's autocoding process. GrTT behavior models may be embedded as equivalent nodes' primitives in an equivalent graph generated during software architecture verification in the team's codesign process. Equivalent graph execution using GrTT behavior models will support validation of application requirements capture through the translation process. Since Ada syntax is used in VHDL, Ada procedures implementing behavior of PGM graph partitions will be common for hardware and software implementations. Because of this, GrTT behavior models of hardware partitions may be embedded as the procedural part of a VHDL behavior architecture, thus

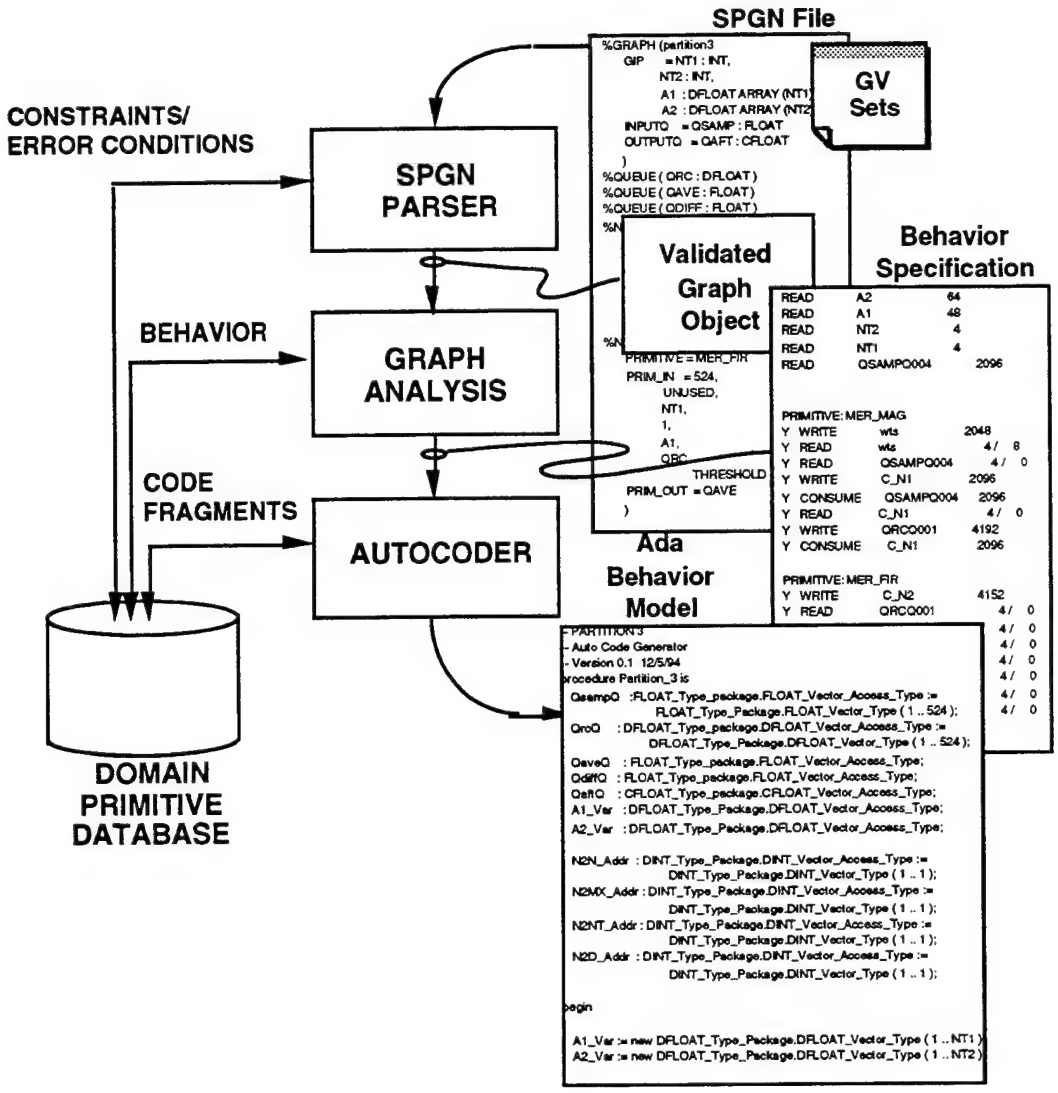


Figure 8. - Graph Translation Tool Architecture and Domain Primitive Database Support. Ada behavior models are autocoded intermediate behavior specifications translated from PGM graphical specifications.

automating generation of VHDL behavior models from graphical architecture specifications.

Figure 9 illustrates the use of a GrTT generated behavior model of a PGM graph partition in validating MPIDs, the target specific partition executables generated by MCCI's autocoding tools. A GrTT behavior model is generated from a partition graph. Input vectors are generated or captured from a higher level algorithm design tool; e.g., PGSE or MATLAB. The behavior model is executed on the captured test vectors and output vectors produced. A test support utility executes the behavior model as the primitive of a single node test application. Input and output vectors are shown above the single node graph GrTT behavior model test application. Input and output vectors are generated

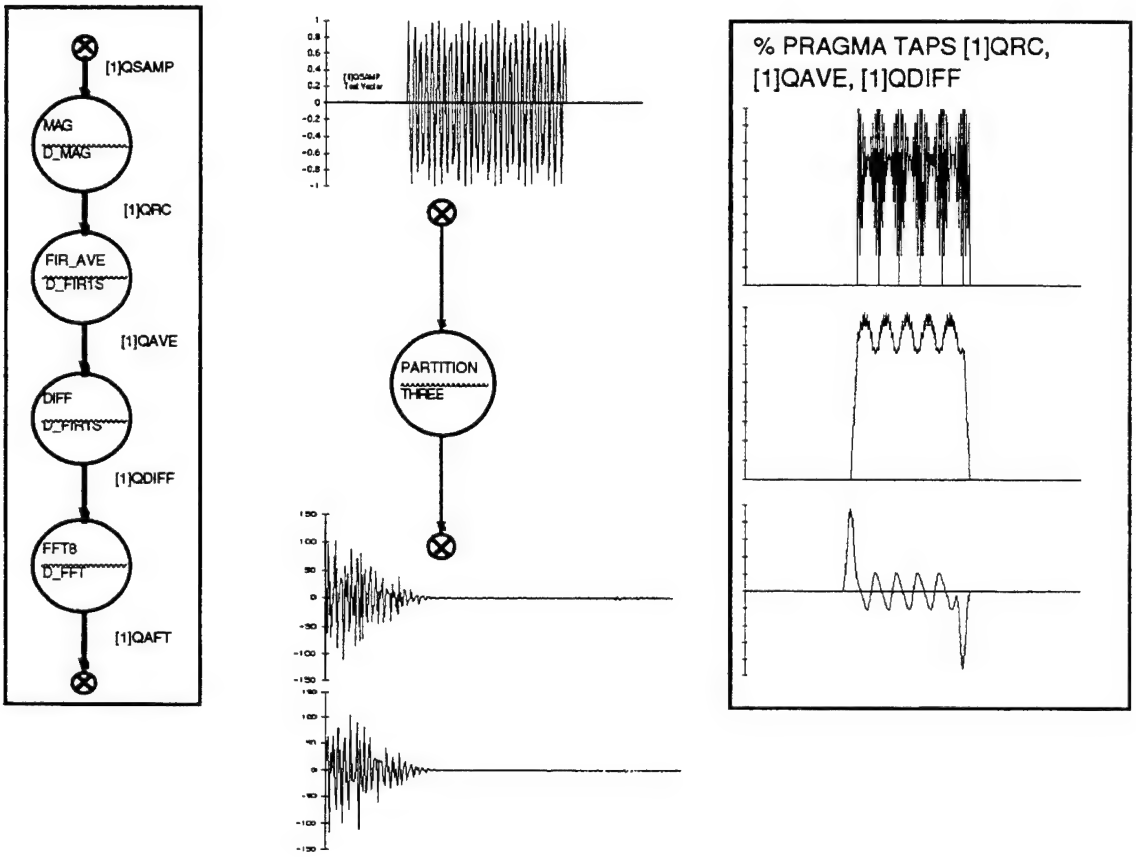


Figure 9 - GrTT Behavior Model Execution
Input and output test vectors and internal queue content
of the behavior model are shown

for use in MPID unit testing. Target specific MPIDS must produce identical output vectors from common input test vectors within precision error limits. The internal partition queue contents are made visible at each stage of behavior model execution by the behavior model test support. This provides the user with a virtual oscilloscope view of internal partition behavior. Internal test vectors are saved and used for debugging target specific MPIDs.

GrTT Concept of Operations

The GrTT tool generates Ada behavior models for DSP application top level design partitions and allocations. Behavior models provide validation of algorithm functional requirements capture by the top level software design. Behavior models are used to validate requirements capture in top level hardware and software partition specifications. Comparison of the behavior models functional behavior with that of the high level functional simulation of the processing algorithms validates the top level specification. The behavior models then form executable behavior requirement specifications for the corresponding partition of the top level design. Behavior requirements specifications embodied in the behavior models are used at successive stages

of autocoding an application. In successive levels of the autocoding process, the behavior of the autocoded unit of an application is compared with that of the behavior model. Execution behavior; i.e., the sequence of primitive executions and the intermediate queue states, must correspond identically with that of the behavior model. Numerical results at each stage of autocoded unit behavior must correspond with the comparable results within precision limits. Applications in which each executable unit has been validated with respect to its behavior model will execute correctly, implementing the behavior that is the union of all model's behavior. If the behavior of each model has been verified as capturing the behavior requirements of the application architecture and all elements of the architecture specification are represented by a behavior model, the end application will correctly and fully implement the requirements.

The concept of operations of GrTT is illustrated by walking through the autocoding of the RASSP Program SAR benchmark. This application is being used by the Lockheed Martin RASSP team as the benchmark to measure tool performance and contribution to achieving program productivity enhancement goals. A graphical specification of a single polarization of the full application was developed by the Lockheed Martin ATL•Camden RASSP prime and distributed to subcontractors for use as a common benchmark for use in developmental testing in the RASSP Program. GrTT is used at each stage of translation of the application architecture specification into executable code for a target architecture. Validation of the translation and verification of requirements capture from the previous stage are illustrated. GrTT's contribution to productivity enhancement is emphasized.

Hardware/Software Architecture Specification. The application is specified as a domain primitive data flow graph using the PGM formalism. The application specification includes both hardware and software allocations. Figure 10 is an illustration of this graph implementing SAR processing on a reduced data set, 256 azimuth resolutions for 64 range cells. Each of the square icons represents a subgraph. The subgraphs are io board, range, and azimuth. Each of the subgraphs are also shown in the figure. The notational form of the specifications using Signal Processing Graph Notation (SPGN) are included in this report as Attachment 1.

The architecture specification tool generated a trial hardware architecture, an allocation of application segments to hardware and software implementations, and processor assignments of nodes within the software allocation. In the benchmark, the io_board subgraph and collect node were allocated to hardware implementations by the architecture specification tool. The range and azimuth subgraphs were allocated to software implementations, with all nodes within a subgraph allocated to the same processor. The architecture specification is input to the autocoding process as a domain primitive PGM graph of the application, a hardware architecture description file, and node assignment lists assigning nodes to either programmable hardware nodes or fixed function hardware nodes. Because the application architecture specification for both hardware and software allocations stems from a common PGM graph, the autocoding process may be used to create rapid software

prototypes of the hardware allocations. These may be later removed from the software architecture when hardware nodes are implemented. This was done for the io_board subgraph and collect node in our example. Behavior models generated with GrTT will be common to both rapid software prototypes and hardware implementations.

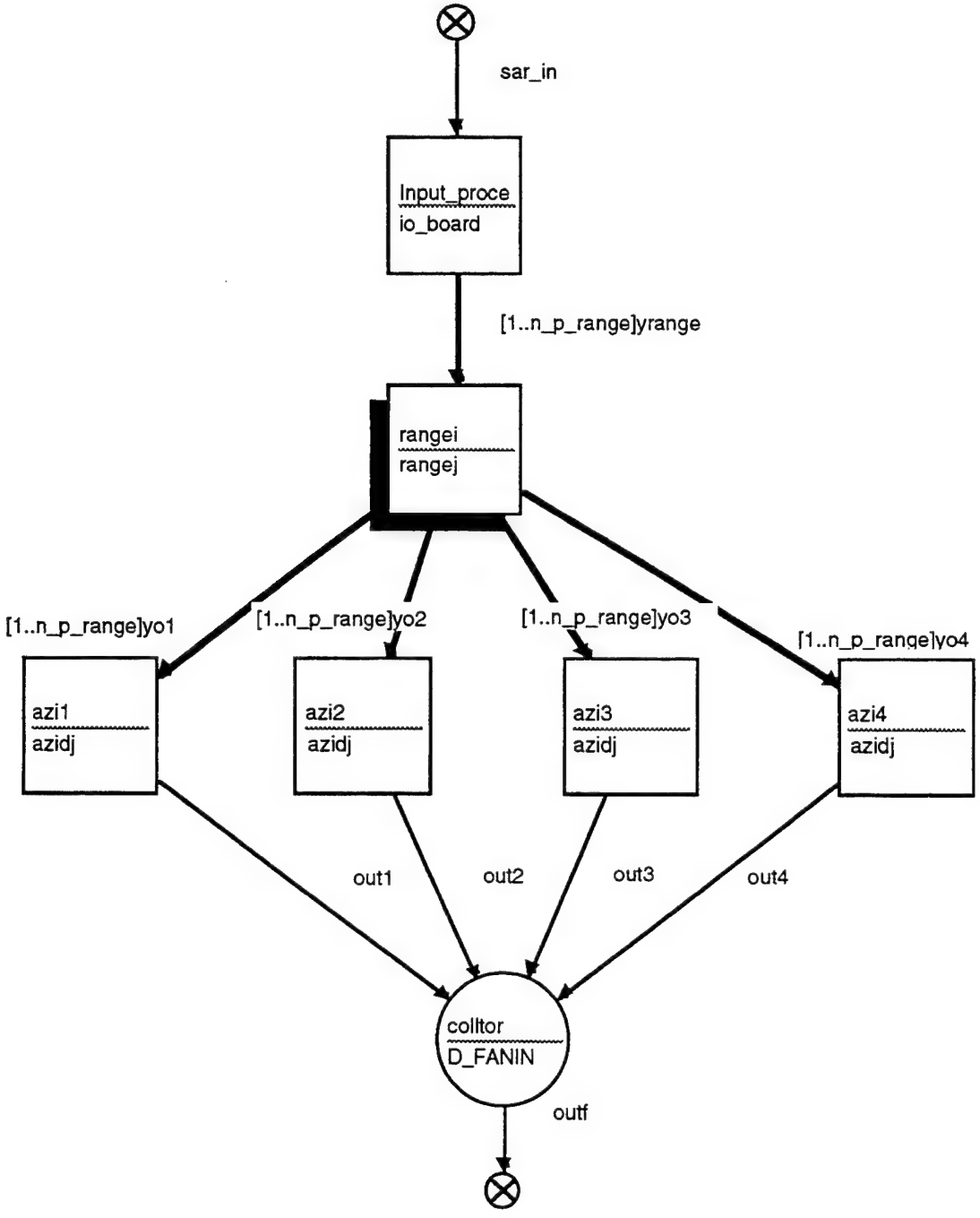


Figure 10. - Application Graph - Domain Primitive Graph for Mini SAR Benchmark

Autocoding Process. MCCI's top level design tool, the Equivalent Application Generator, is used to generate allocation, partition, and equivalent application graphs from the application architecture graph. The Equivalent Application Generator tool is the first of the set of autocoding tools. This tool creates stand-alone graphs from specified clusters within input PGM graphs. Nodes in an input graph may belong to a single partition only and all nodes must belong to a partition. Partitions may be belong to either hardware or software allocations. Partitions are defined to the tool by listing the member nodes in each partition in a pragma contained in the notational form of the graph. By listing each hardware partition's nodes and all the remaining nodes in the software allocation as a single partition, hardware partition and the full software allocation graphs are created as partition graphs. This initial process separates the hardware and software allocations into separate graphs that now follow parallel paths through the hardware/software development paths of the codesign process. Figure 11 shows the software allocation, the software architecture graph, and the hardware partition graphs. The hardware partition graphs for io_board and collect and the software allocation graph in SPGN form are included as Attachment 2.

GrTT behavior models are created for each of the hardware partitions. Hardware partition behavior models may be used as executable behavior models for the hardware design process. Because Ada syntax and VHDL syntax are essentially identical, the procedure part of the Ad behavior model may be readily incorporated as the procedural body of a VHDL behavior model. The differences between VHDL behavior models and GrTT behavior models are in the declarations. The stand-alone test support may be used to execute the GrTT Ada behavior model on input test vectors taken from the algorithm functional simulation. Output vectors that match functional simulation vectors within precision limits validate requirements capture in the PGM hardware partition specification. The GrTT behavior model and test support may be used in subsequent stages of hardware design to validate hardware implementation. Test vectors generated by the model may be used at lower levels of hardware development testing. The behavior model generated by GrTT for io_board hardware partition graph is attached as Attachment 3.

The software architecture graph is partitioned for assignment to the processors of the specified hardware architecture. Partitioning implements the processor assignments received from the architecture specification tool as one or more software partitions on the specified processor. In the benchmark, the assignment lists and partition lists correspond with the range and azimuth subgraphs. Stand-alone partition graphs are generated for each partition specified. An equivalent application graph is generated in which each partition is replaced by a single equivalent node. The equivalent node represents a DSP program that implements partition behavior on the target DSP processor. The GrTT behavior models will model that behavior. Figure 12 illustrates the software allocation's partition graphs and the equivalent application graph. The SPGN form of these graphs is included as Attachment 4.

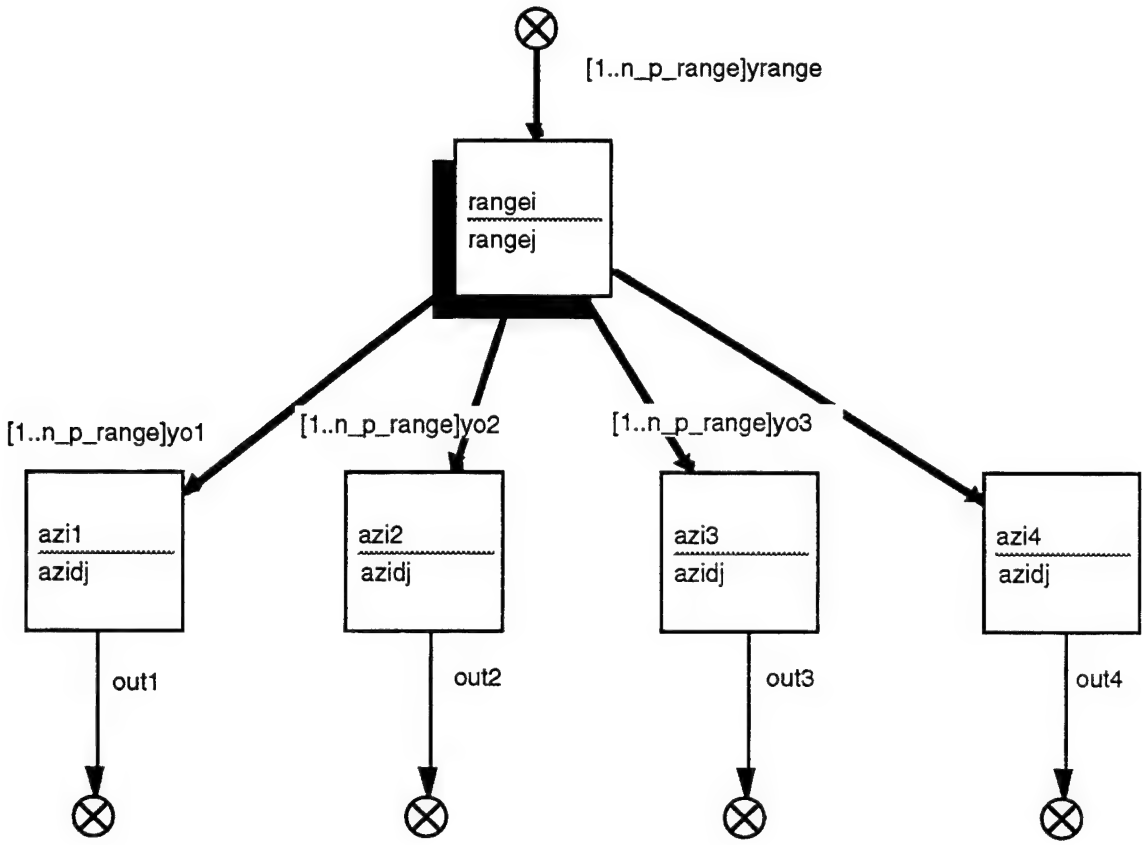


Figure 11a. - Software Allocation Graph - Mini SAR Benchmark

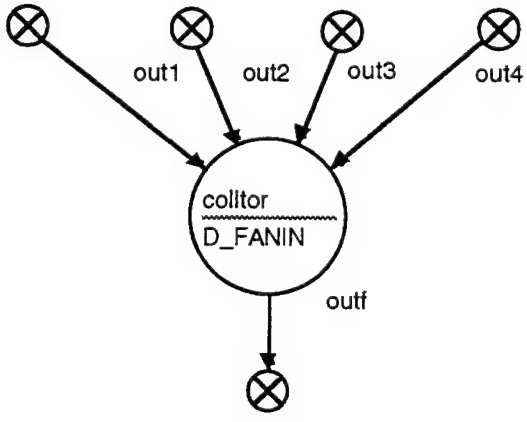


Figure 11b. - Hardware Partitions - Mini SAR Benchmark

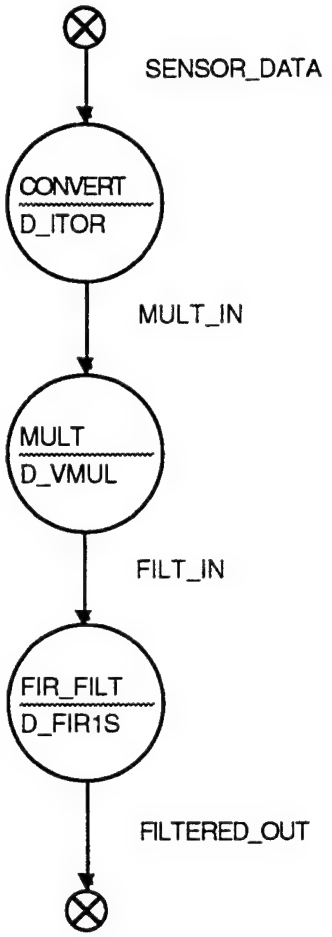


Figure 11b. - (cont.) Hardware Partitions - Mini SAR Benchmark

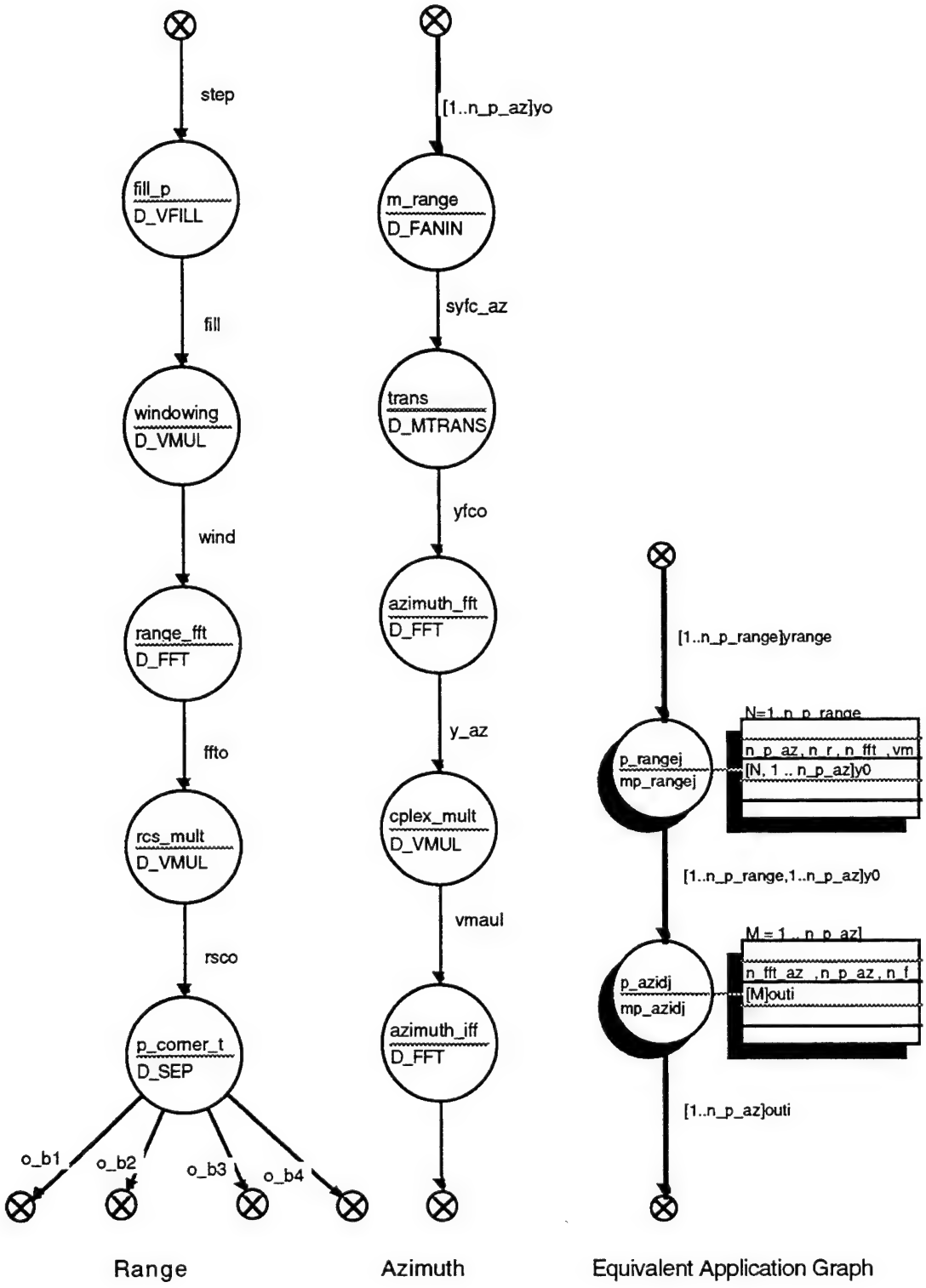


Figure 12. - Software Partition Graphs - Mini SAR Benchmark

GrTT is used to create behavior models of each software partition. Partition graphs and a file of the graph value sets, the sets of enumerated values of

controls for which GrTT is to create a valid behavior model are input to the GrTT tool for each partition. Behavior models are generated. The behavior models are executed under the test support utility using test vectors generated by the functional simulator. Comparison within precision limits verifies requirements capture by the partition specification and validates the model for use in target specific MPID unit testing.

The use of GrTT to generate behavior models is illustrated in a walk through of the translation process using the range partition graph as an example. Required inputs to GrTT are the SPGN for the partition graph and a partition specific Graph Value Set. The partition graph was generated from the software allocation graph by the Equivalent Application Builder tool in SPGN notation. As previously mentioned, the SPGN for this example is included in Attachment 1. Since no values for any parameters are required to perform the translation, the Graph Value Set is empty (as shown in Attachment 1). The user enters the SPGN file name and the Graph Value Set name on the command line that invokes the GrTT tool. Details of the command line parameters are contained in the GrTT User's Manual.

GrTT parses the SPGN representation of the graph into a set of data structures. This first set of data structures represent a graph realization. Using the values for required parameters as specified in the Graph Value Set, GrTT then creates a set of data structures for each set specified in the Graph Value Set. Each of these sets represent a graph instantiation, one instance of the graph realization. Each graph instance is separately analyzed and a valid node execution sequence is determined for the transient (if any) and periodic execution behavior of the graph. The execution sequences of the graph instances are combined into a single representation that contains logic to select the particular instance that is to be executed based on the values of parameters that affect the execution sequence and therefore requires a different instantiation of the graph. Once this has been performed, the autocoding subsystem of GrTT generates an Ada Specification and an Ada Body that implement the execution sequence(s) of the graph instantiation(s). The code contains a call to the Ada procedure that implements the domain primitive referenced by the node in the execution sequence, code that selects the appropriate execution sequence for each set specified in the Graph Value Set, code that manages the data buffers that implement queues between nodes, and glue logic.

For the range partition, there is only one execution sequence and that is simple in that each node executes once. The Domain Primitive execution sequence is: D_FILL, D_VMUL, D_FFT, D_VMUL, D_FANOUT. This can be readily seen by examining the Ada generated by GrTT (which has been included in Attachment 5).

Restrictions on graphs that are to be translated into Ada source code are minimal. The major restrictions are:

- The input graph must be balanced.

• At graph translation time, there must be sufficient information to determine an execution sequence, and the amount of data produced and consumed by each node must be known. Values for formal GIPs and VARs, which are normally provided at instantiation time, will be provided at graph translation time so that these restrictions are met. This means that a graph's execution sequence cannot be dependent on run-time data.

A secondary output of GrTT is an interface with a test environment such that the GrTT produced procedure can be tested. The test environment contains facilities to read input queue data from a file and to write output queue data to a file.

The Ada behavior models generated by GrTT for the range and azimuth partition graphs are included as Attachment 5. The Ada test programs generated by GrTT are included as Attachment 6.

The behavior model provides both a static and a functional model of each partition's execution behavior that may be used to validate requirements capture by the partition graph specification. It also provides a validated behavior model that may be used in support of unit testing the target specific partition translations. Figure 13 illustrates the behavior of a partition graph that the model captures. For each set of graph values, an execution sequence of domain primitive executions is generated. The sequence will be periodic, but may also have an initial transient sequence. Between each domain primitive execution, the partition graph state is modeled as the the contents of circular buffers implementing the queues. The amount of valid data and its location within the buffer models the contents of a queue. The contents of all queues define the graph state. The execution of a primitive transitions the graph to its next state. This state machine behavior of the partition graphs will be common to all target specific translations regardless of the target processor or supporting vendor primitive library used to implement domain primitive specifications. Values of the data at each graph state modeled may be used for functional testing of target specific translations. Figure 13 shows execution of the range_fft node transitioning the behavior model to its third active state, the contents of buffer ffto filled with active data denoting its third state, and a plot of the contents of the ffto buffer modeling the queue contents of queue ffto after range_fft firing. GrTT currently uses a feature called "taps" to extract and display the contents of internal queues that are converted to circular buffers. This feature is simular to virtual pins used in VHDL hardware modeling. The queues to be "tapped" are specified in a pragma in the input SPGN. Tapped internal queue contents are output as a fromal output. The internal queue "ffto" was tapped in this example. In the planned upgrade to GrTT that will integrate it with the autocode toolset unit tester, all internal queue contents will be automatically retrieved and plotted during model exection.

Each partition graph with graph value sets is autocoded into source code for its target architecture processor using the MPID generator tool. Target specific translations implement the partition graph behavior for each graph value set utilizing the primitives from an optimized math library supporting the processor.

One or more target specific math library primitives is required to implement a domain primitive. Execution behavior generated in the translation processor as documentation may be compared with the behavior of the model generated by GrTT. Execution sequences and queue states must be common. A test image

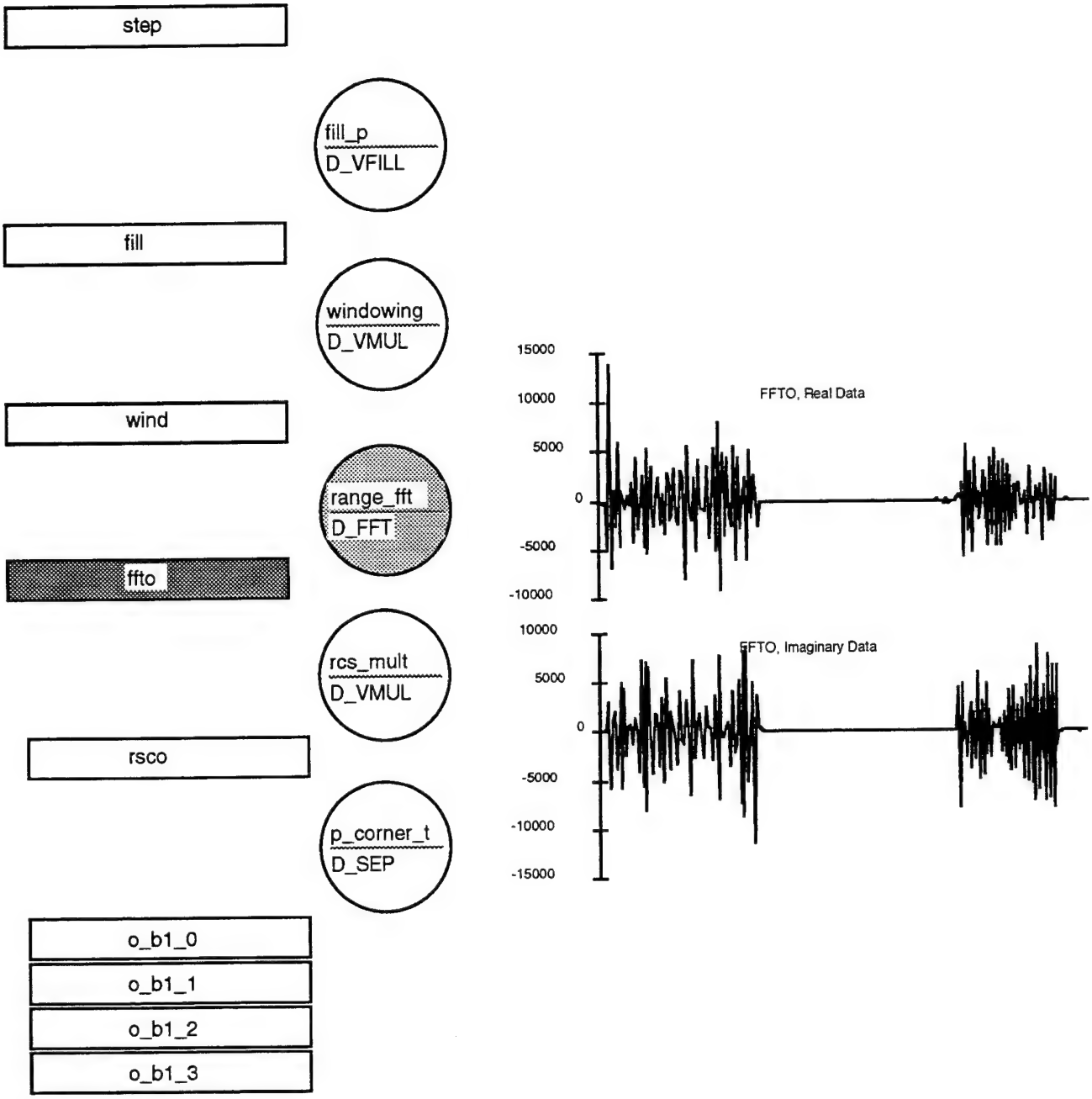


Figure 13. - Range Software Partition Graph Behavior Model - Mini SAR Benchmark

is automatically generated for unit validation testing. This is a single node application with the MPID as its primitive. A unit tester is used to test the MPID. This utility executes the single node on a file of test vectors specified by the user. Queue contents are reconstructed from the memory map of the MPID at each execution state. Comparison with the queue contents of the MPID with the corresponding queue state of the behavior model will quickly validate the target specific translation. The GrTT stand-alone test utility and the MPID unit tester will be integrated in a future release of the autocoding tools supporting side-by-side execution of the behavior model and test image. Figure 14 illustrates the output of MPID unit testing and the comparison of MPID output vectors with GrTT behavior model vectors generated by the GrTT Test Utility .

The Application Generator tool is used to specify a complete software system to the compiler for the target architecture's programmable processors. The equivalent application graph SPGN file, MPID source files, and hardware description file are translated into an application configuration file. The configuration file is the run-time image used by the run-time support system to create an instantiation of the application. A make file specifying the software system to the target compiler is generated. The file contains the configuration file, MPID source files, primitive object code, operating system, and BIT applications. In test and integration, functional behavior of the application is verified by executing the graph on test vectors used in earlier stages of the autocoding process. Comparison of the application's formal queue contents with the output generated by GrTT behavior models validates the end product of the autocoding process.

A major element in achieving a 4x productivity improvement in the RASSP process is following a correct by construction software development methodology which is embodied in the Lockheed Martin RASSP design methodology. Radical reduction of TAF, test and fix, activity in hardware development, and software debugging in software implementation are a major part of achieving these goals. Elimination of this often very expensive fixing up activity at the end of a codesign effort is to be achieved by several intermediate stages of verifying requirements capture at each level of abstraction in the codesign process and validating the correctness of the realization of the design at that level. If rigorously followed, this "correct by construction" method will produce software that works the first time it is executed, hardware that functions as intended in initial testing, and full compatibility between hardware and software realizations of the codesign. GrTT provides behavior models that support users meeting the correct by construction goal by verifying requirements capture and validating the products of the translation process.

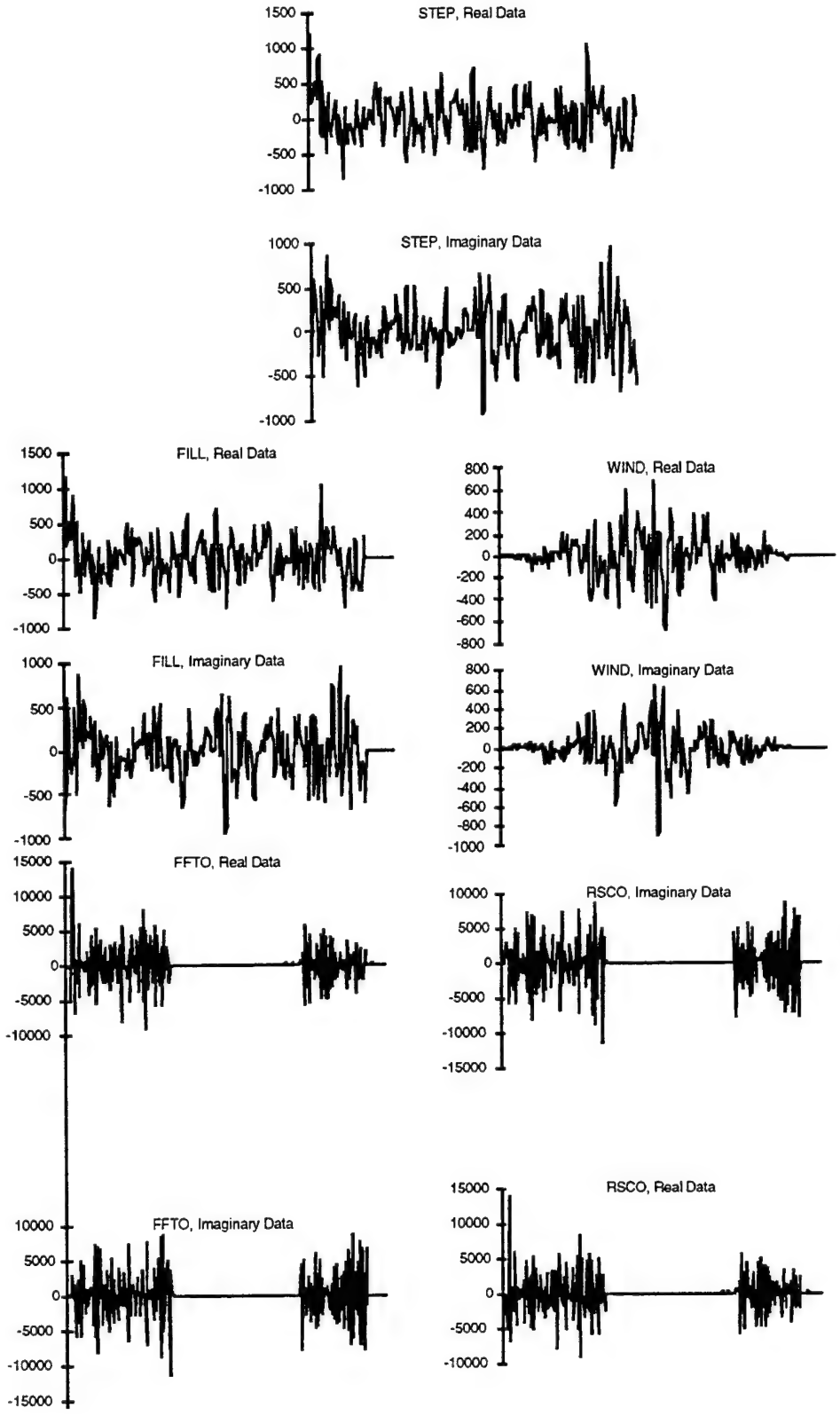


Figure 14. - Input and Internal Queue Contents from MPID Unit Testing - Mini SAR Benchmark

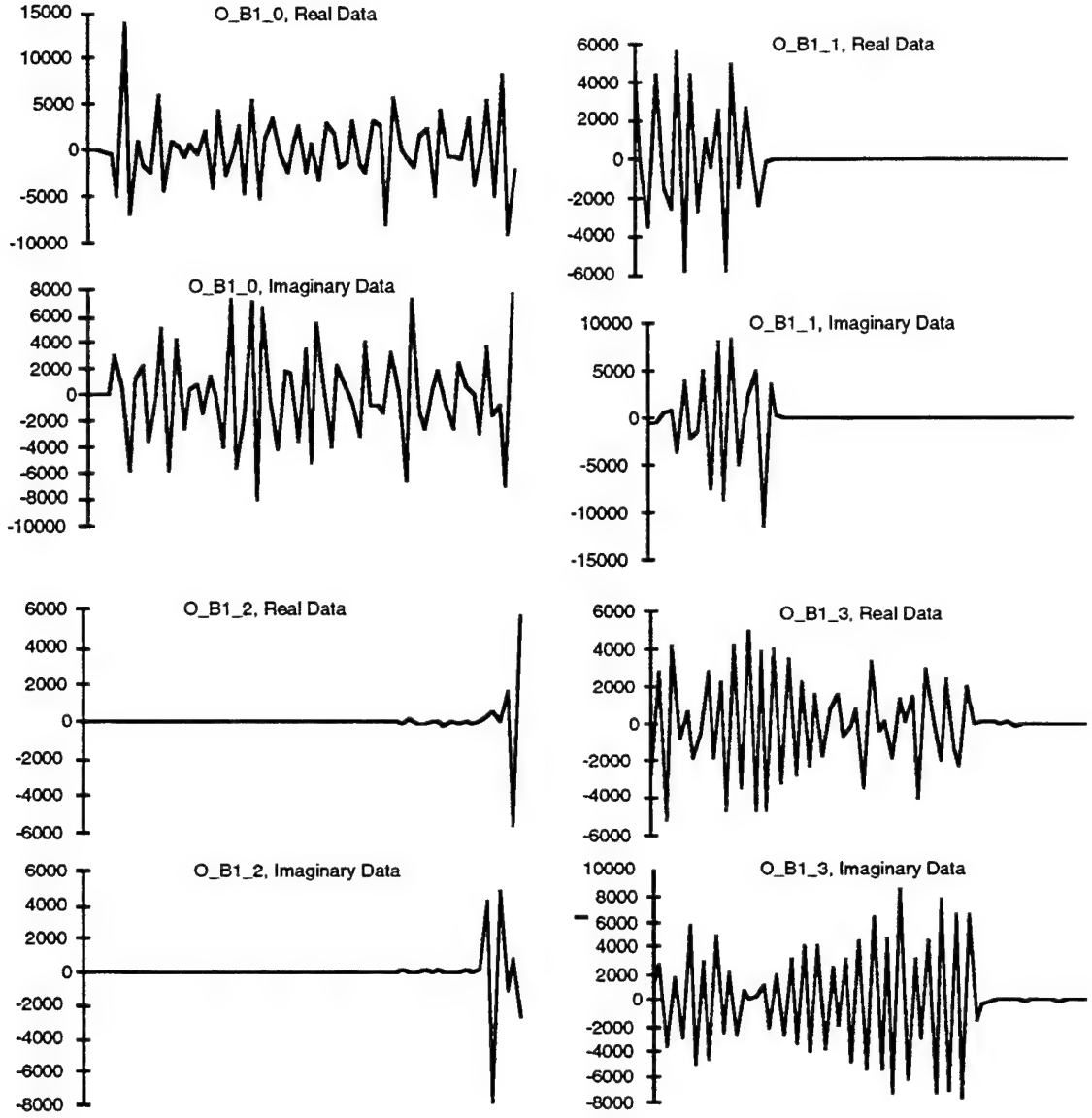


Figure 14. - (cont.) Output Queue Contents from MPID Unit Testing - Mini SAR Benchmark

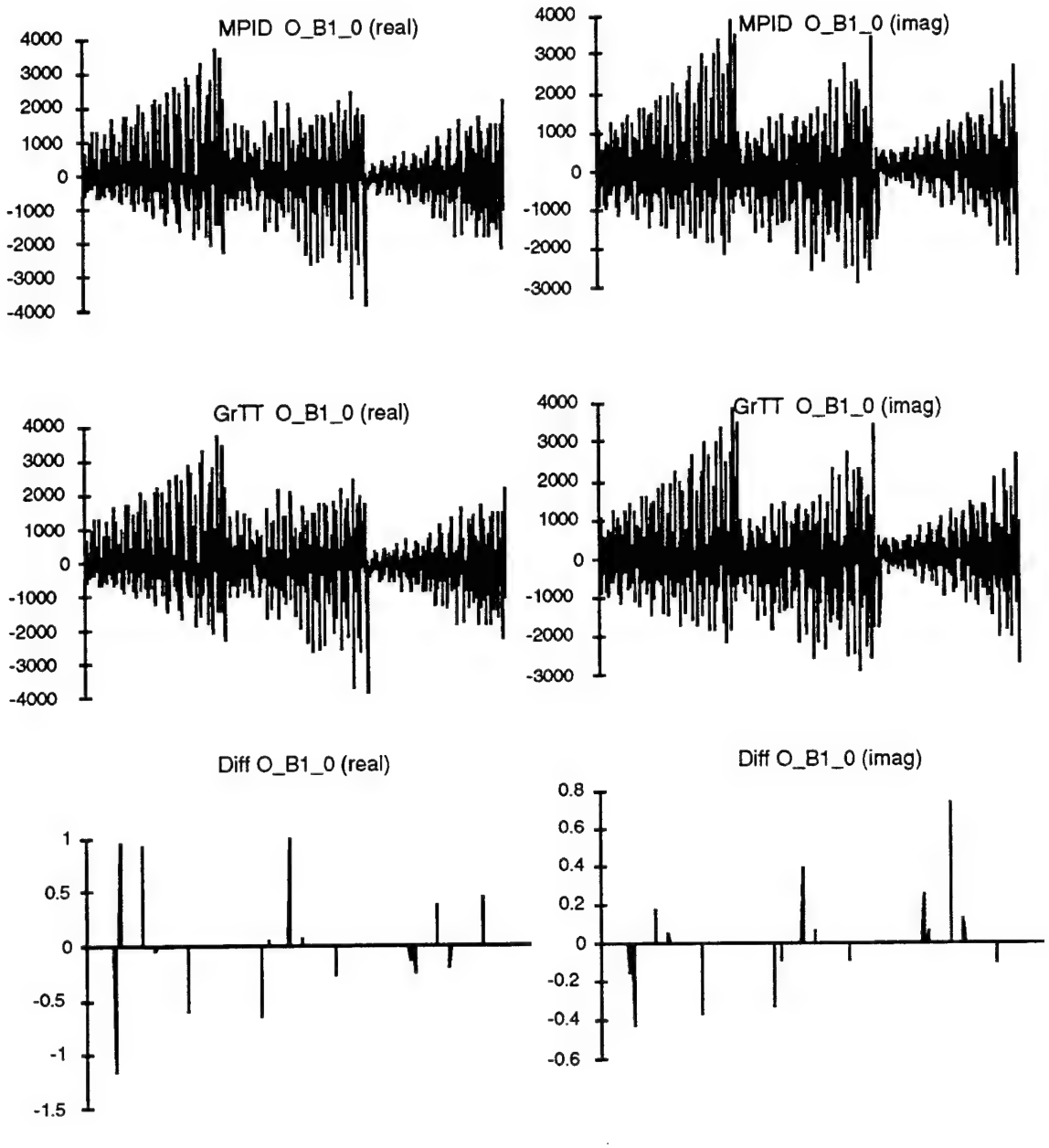


Figure 14. - (cont.) Comparison of Output Queue Contents from MPID Unit Testing and GrTT Test Utility - Mini SAR Benchmark

ATTACHMENT 1
SPGN For Range and Azimuth Partitions

Range Example

The Range example is taken directly from a Synthetic Aperature Radar (SAR) application that has been used as a benchmark in the RASSP Program. The Range graph implements the front end processing after baseband conversion and filtering have been performed. The processing incorporates pulse compression. In-phase and quadrature samples are first weighted to reduce the sidelobe structure of the compressed pulse and to compensate for the non-ideal IF filter characteristics. Weighted I/Q data are transformed to compressed range data using a Fourier Transform. Compensation occurs after the pulse compression to account for radar cross section variations due to elevation beam-shape modulation and R^4 losses.

SPGN for Range Graph

```
%GRAPH (P_RANGE
       GIP       = VMUL : CFLOAT ARRAY(256),
                   RCSMUL : FLOAT ARRAY(256)
       INPUTQ    = STEP : CFLOAT
       OUTPUTQ   = O_B1 : CFLOAT,
                   O_B2 : CFLOAT,
                   O_B3 : CFLOAT,
                   O_B4 : CFLOAT
       )

%GIP ( N_R : INT      INITIALIZE TO 470)
%GIP ( N_P_AZ : INT   INITIALIZE TO 4)
%GIP ( N_FFT : INT    INITIALIZE TO 256)
%GIP (PAD : CFLOAT    INITIALIZE TO <0.0E0, 0.0E0> )
%GIP (P : DINT ARRAY (4, 1)
                   INITIALIZE TO { 64,
                                   64,
                                   64,
                                   64 } )

%QUEUE ( FILL : CFLOAT )
%QUEUE ( WIND : CFLOAT )
%QUEUE ( FFTO : CFLOAT )
%QUEUE ( RSCO : CFLOAT )

%NODE ( FILL_P
       PRIMITIVE = D_VFILL
       PRIM_IN   = N_R/2,
                   N_FFT - N_R/2,
                   0,
                   PAD,
                   STEP
                               THRESHOLD = N_R/2
       PRIM_OUT  = FILL
       )
```

```
%NODE ( WINDOWING
        PRIMITIVE = D_VMUL
        PRIM_IN   = N_FFT,
                    UNUSED,
                    FILL
                                THRESHOLD = N_FFT,
                    VMUL
        PRIM_OUT  = WIND
        )

%NODE ( RANGE_FFT
        PRIMITIVE = D_FFT
        PRIM_IN   = N_FFT,
                    N_FFT,
                    0,
                    1,
                    UNUSED,
                    WIND
                                THRESHOLD = N_FFT
        PRIM_OUT  = FFTO
        )

%NODE ( RCS_MULT
        PRIMITIVE = D_VMUL
        PRIM_IN   = N_FFT,
                    UNUSED,
                    FFTO
                                THRESHOLD = N_FFT,
                    RCSMUL
        PRIM_OUT  = RSCO
        )

%NODE ( P_CORNER_T
        PRIMITIVE = D_FANOUT
        PRIM_IN   = N_FFT,
                    N_P_AZ,
                    P,
                    1,
                    RSCO
                                THRESHOLD = N_FFT
        PRIM_OUT  = FAMILY [O_B1,O_B2,O_B3,O_B4],
                    UNUSED

        )
%ENDGRAPH
```

Graph Value Set for Range Graph

Since there are no graph variables that require values in order for the translation to occur, the Graph Value Set is empty.

```
%GV_SET
%END_SET
```

Azimuth Example

The Azimuth example is taken directly from a Synthetic Aperature Radar (SAR) application that has been used as a benchmark in the RASSP Program. The Azimuth graph implements the processing after range processing has been performed. The processing incorporates cross-range convolution filtering. Compressed range pulses are placed in time sequence into a two-dimensional array and each row of the array is convolved with a row specific kernel. The convolution processing is performed using FFTs with the overlap and save method.

SPGN for Azimuth Graph

```
%GRAPH (P_AZI
       INPUTQ    = [1..4]YO : CFLOAT,
                   VMUL_AZ : CFLOAT ARRAY (128)
       OUTPUTQ   = YOA_AZ : CFLOAT
       )

%GIP (N_P_AZ : INT INITIALIZE TO 4)
%GIP (N_F_A : INT  INITIALIZE TO 64)
%GIP (N_FFT_AZ : INT INITIALIZE TO 128)
%GIP (N_FFT_AZ2 : INT INITIALIZE TO N_FFT_AZ/2 )
%GIP (PAZ : DINT ARRAY (4, 1)
                   INITIALIZE TO { 64,
                                   64,
                                   64,
                                   64 } )

%QUEUE ( SYFC_AZ : CFLOAT )
%QUEUE ( YFCO : CFLOAT )
%QUEUE ( Y_AZ : CFLOAT )
%QUEUE ( VMAUL : CFLOAT )

%NODE ( M_RANGE
       PRIMITIVE = D_FANIN
       PRIM_IN   = N_F_A  * N_P_AZ,
                   N_P_AZ,
                   PAZ,
                   1,
                   [1..4]YO
                            THRESHOLD = N_F_A*N_FFT_AZ2 / 2
                            CONSUME   = N_F_A*N_FFT_AZ2/N_P_AZ
       PRIM_OUT  = SYFC_AZ,
                   UNUSED
       )

%NODE ( TRANS
       PRIMITIVE = D_MTRANS
       PRIM_IN   = N_FFT_AZ,
                    N_F_A,
                    SYFC_AZ
                       THRESHOLD = N_F_A*N_FFT_AZ
       PRIM_OUT  = YFCO
       )

%NODE ( AZIMUTH_FFT
```

```
        PRIMITIVE = D_FFT
        PRIM_IN   = N_FFT_AZ,
                    N_FFT_AZ,
                    0,
                    1,
                    UNUSED,
                    YFCO
                              THRESHOLD = N_F_A*N_FFT_AZ
                              %%THRESHOLD = N_F_A*N_FFT_AZ/4
        PRIM_OUT  = Y_AZ
        )

%NODE ( CPLEX_MULT
        PRIMITIVE = D_VMUL
        PRIM_IN   = N_FFT_AZ,
                    UNUSED,
                    Y_AZ
                              THRESHOLD = N_F_A*N_FFT_AZ,
                    VMUL_AZ
                              THRESHOLD = 1
        PRIM_OUT  = VMAUL
        )

%NODE ( AZIMUTH_IFFT
        PRIMITIVE = D_FFT
        PRIM_IN   = N_FFT_AZ,
                    N_FFT_AZ2,
                    1,
                    N_FFT_AZ2+1,
                    UNUSED,
                    VMAUL
                              THRESHOLD = N_F_A*N_FFT_AZ
        PRIM_OUT  = YOA_AZ
        )
%ENDGRAPH
```

Graph Value Set for Azimuth Graph

Since there are no graph variables that require values in order for the translation to occur, the Graph Value Set is empty.

```
%GV_SET
%END_SET
```

ATTACHMENT 2
SPGN for the IO_BOARD Hardware Partition Graph

```
%GRAPH(IO_BOARD
    GIP = NTAPS      : INT,
          FILT_WTS   :FLOAT ARRAY (NTAPS)
    INPUTQ = SENSOR_DATA     : CINT
    OUTPUTQ = FILTERED_DATA : CFLOAT
    )

%GIP (NPTS : INT
          INITIALIZE TO 236)

%GIP (MULT_ARRAY : CFLOAT ARRAY(2)
                     INITIALIZE TO {<-1.0E0, -1.0E0>, <1.0E0, 1.0E0>} )

%QUEUE (MULT_IN : CFLOAT)
%QUEUE (FILT_IN : CFLOAT)

%NODE(CONVERT
    PRIMITIVE = D_ITOR
    PRIM_IN = NPTS,
              UNUSED,
              UNUSED,
              SENSOR_DATA
                  THRESHOLD = NPTS
    PRIM_OUT = MULT_IN
    )

%NODE(MULT
    PRIMITIVE =D_VMUL
    PRIM_IN = NPTS,
              UNUSED,
              MULT_IN
                  THRESHOLD = NPTS,
              MULT_ARRAY
    PRIM_OUT = FILT_IN
    )

%NODE(FIR_FILT
    PRIMITIVE = D_FIR1S
    PRIM_IN = NPTS,
              UNUSED,
              NTAPS,
              UNUSED,
              FILT_WTS,
              FILT_IN
                  THRESHOLD = NPTS
    PRIM_OUT =  FILTERED_DATA
    )

%ENDGRAPH
```

ATTACHMENT 3
IO_BOARD Behavior Model

Ada Specification for IO_BOARD

```
--+--------------------------------------------------------------------------
--| File: io_board_.ada
--| Generated by the MCCI Graph Translation Tool (GrTT) - Version: 1.0
--| On 04/25/96, at 16:07:56
--+--------------------------------------------------------------------------
--|
--| With Clauses for Basic GrTT Types
with Cfloat_Type_Package;
with Cint_Type_Package;
with Float_Type_Package;
with Int_Type_Package;

package Io_Board is
   --|
   --| Procedure: Init
   --| Used to initialize all queues in the GrTT PID.  If the
   --| procedure Pid is called before this procedure, the exception
   --| Uninitialized_Pid shall be raised.
   --|
   procedure Init (
      Ntaps : in Natural;
      Filt_Wts : in Natural;
      Sensor_Data : in Natural;
      Filtered_Data : in Natural);
   --|
   Uninitialized_Pid : exception;
   --|
   --|
   --| Procedure: Pid
   --| This procedure shall perform the same functionality as the
   --| input partition graph Io_Board.
   --|
   procedure Pid (
      Ntaps : in out Int_Type_Package.Int_Vector_Access_Type;
      Filt_Wts : in out Float_Type_Package.Float_Vector_Access_Type;
      Sensor_Data : in out Cint_Type_Package.Cint_Vector_Access_Type;
      Filtered_Data : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type);
   --|
end Io_Board;
```

Ada Body for IO_BOARD

```
--+--------------------------------------------------------------------------
--| File: io_board.ada
--| Generated by the MCCI Graph Translation Tool (GrTT) - Version: 1.0
--| On 04/25/96, at 16:07:56
--+--------------------------------------------------------------------------
--|
--| With Clauses for Basic GrTT Types
with Cfloat_Type_Package;
```

```
with Float_Type_Package;
with Dfloat_Type_Package;
with Int_Type_Package;
--|
--| Use Clauses for Basic GrTT Types
use Float_Type_Package;
use Int_Type_Package;
--|
--| With Clauses for GrTT Queue Managers
with Cfloat_Queue_Manager;
with Cint_Queue_Manager;

--| With Clauses for GrTT Algorithms
with D_Fir1s;
with D_Itor;
with D_Vmul;

package body Io_Board is
   --|
   --| GrTT PID type definitions...
   --|
   type Gv_State_Type is range 0..1;
   type Comp_State_Type is range 0..1;
   --|
   --|
   --| GrTT PID Constant & Variable definitions...
   --|
   --| PID State Constants & Variables:
   --|
   Top_Of_Period : constant Gv_State_Type := Gv_State_Type'First;
   Composite_Init : constant Comp_State_Type := Comp_State_Type'First;
   --|
   Gv_State : Gv_State_Type;
   Comp_State : Comp_State_Type;
   --|
   --| Declare Graph Variables...
   Npts : Int_Type_Package.Int;
   Mult_Array_Entity : Cfloat_Type_Package.Cfloat_Vector_Type (0..1);
   Mult_Array : Cfloat_Type_Package.Cfloat_Vector_Access_Type
      := new Cfloat_Type_Package.Cfloat_Vector_Type'(Mult_Array_Entity);
   --|
   --| Declare Formal Queues...
   QMult_In : Cfloat_Queue_Manager.Queue_Type;
   QFilt_In : Cfloat_Queue_Manager.Queue_Type;
   QSensor_Data : Cint_Queue_Manager.Queue_Type;
   QFiltered_Data : Cfloat_Queue_Manager.Queue_Type;
   --|
   --| Initialization Flag Declaration...
   Initialized : Boolean := False;
   --|
   --|
   function Determine_Gv_Set return Gv_State_Type is
   begin
      return 1;
   end Determine_Gv_Set;
   --|
   --|
   procedure Init (
```

```
   Ntaps : in Natural;
   Filt_Wts : in Natural;
   Sensor_Data : in Natural;
   Filtered_Data : in Natural) is
begin
   --| Initialize State Variables:
   Gv_State := Top_Of_Period;
   Comp_State := Composite_Init;
   --|
   --| Set Initialize Flag
   Initialized := True;
   --|
   --| Initialize Formal Queue Buffers
   Cint_Queue_Manager.Initialize (Sensor_Data, 0, QSensor_Data);
   Cfloat_Queue_Manager.Initialize (Filtered_Data, 0, QFiltered_Data);
   --|
   --| Initialize Persistent Queue Buffers (if any)
end Init;
--|
--|
procedure Pid (
   Ntaps : in out Int_Type_Package.Int_Vector_Access_Type;
   Filt_Wts : in out Float_Type_Package.Float_Vector_Access_Type;
   Sensor_Data : in out Cint_Type_Package.Cint_Vector_Access_Type;
   Filtered_Data : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type)
   is

   procedure Io_Board_Kernel (
      QSensor_Data : in out Cint_Queue_Manager.Queue_Type;
      QFiltered_Data : in out Cfloat_Queue_Manager.Queue_Type) is
      --|
      --| Local Data Vector Declarations
      --|
      Mult_In : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
      Filt_In : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   begin
      case Gv_State is
         when Top_Of_Period =>
            Gv_State := Determine_Gv_Set;
            Comp_State := Composite_Init;

         --| Graph Value Set #1
         when 1 =>
            Npts := 236;
            Mult_Array (0).Re := -1.00000000000000E+00
            ;
            Mult_Array (0).Im := -1.00000000000000E+00
            ;
            Mult_Array (1).Re := 1.00000000000000E+00
            ;
            Mult_Array (1).Im := 1.00000000000000E+00
            ;
            case Comp_State is
               --| Initialization Composite State
               when Composite_Init =>
                  --| Init Formal Thresholds
                  Cint_Queue_Manager.Set_Threshold (236, QSensor_Data);
                  --|
```

```
                  --| State Variable Maintenance
                  Comp_State := Comp_State + 1;

               --| Periodic Composite #1
               when 1 =>
                  --| Circular Buffer Inits
                  Cfloat_Queue_Manager.Initialize (236, 236, QMult_In);
                  Cfloat_Queue_Manager.Initialize (236, 236, QFilt_In);
                  --|
                  --| Pode Executions
                  Mult_In := new
                     Cfloat_Type_Package.Cfloat_Vector_Type (0..235);
                  Sensor_Data := new Cint_Type_Package.Cint_Vector_Type'(
                     Cint_Queue_Manager.Read (236, 0, QSensor_Data));
                  D_ITOR.Prim (
                     Dfloat_Type_Package.Compound_Type_Pkg.Unused,
                     Dfloat_Type_Package.Compound_Type_Pkg.Unused, 236,
                     Sensor_Data, Mult_In);
                  Cint_Queue_Manager.Consume (236, QSensor_Data);
                  Cint_Type_Package.Compound_Type_Pkg.Free (Sensor_Data);
                  Cfloat_Queue_Manager.Produce (Mult_In.all, QMult_In);
                  Cfloat_Type_Package.Compound_Type_Pkg.Free (Mult_In);
                  --|
                  Filt_In := new
                     Cfloat_Type_Package.Cfloat_Vector_Type (0..235);
                  Mult_In := new Cfloat_Type_Package.Cfloat_Vector_Type'(
                     Cfloat_Queue_Manager.Read (236, 0, QMult_In));
                  D_VMUL.Prim (
                     236, 3, 236, Mult_In, 2, Mult_Array, Filt_In);
                  Cfloat_Queue_Manager.Consume (236, QMult_In);
                  Cfloat_Type_Package.Compound_Type_Pkg.Free (Mult_In);
                  Cfloat_Queue_Manager.Produce (Filt_In.all, QFilt_In);
                  Cfloat_Type_Package.Compound_Type_Pkg.Free (Filt_In);
                  --|
                  Filtered_Data := new
                     Cfloat_Type_Package.Cfloat_Vector_Type (0..228);
                  Filt_In := new Cfloat_Type_Package.Cfloat_Vector_Type'(
                     Cfloat_Queue_Manager.Read (236, 0, QFilt_In));
                  D_FIR1S.Prim (
                     236, 1, 8, 1, Filt_Wts, 236, Filt_In, Filtered_Data);
                  Cfloat_Queue_Manager.Consume (236, QFilt_In);
                  Cfloat_Type_Package.Compound_Type_Pkg.Free (Filt_In);
                  Cfloat_Queue_Manager.Produce (Filtered_Data.all,
                  QFiltered_Data);
                  Cfloat_Type_Package.Compound_Type_Pkg.Free (
                  Filtered_Data);
                  --|
                  --| Formal Threshold Updates
                  Cint_Queue_Manager.Set_Threshold (0, QSensor_Data);
                  --|
                  --| State Variable Maintenance
                  Gv_State := Top_Of_Period;
            end case;
      end case;
   end Io_Board_Kernel;

begin    --| Beginning of Procedure Pid
   --|
```

```
      --| Verify that the PID's Queues are initialized
      --|
      if not Initialized then
         raise Uninitialized_Pid;
      end if;
      --|
      --|
      --| Move input data from arrays to queues (circular buffers).
      Cint_Queue_Manager.Produce (Sensor_Data.all, QSensor_Data);
      --|
      --| Call PID Kernel Processing.
      while (
         Cint_Queue_Manager.Over_Threshold (QSensor_Data)) loop
         --|
         Io_Board_Kernel (
            QSensor_Data,
            QFiltered_Data);
      end loop;
      --|
      --|
      --| Move the output data from the PID's output queues
      --| onto arrays which output the data to the I/O wrapper.
      --|
      Filtered_Data := new Cfloat_Type_Package.Cfloat_Vector_Type'(
         Cfloat_Queue_Manager.Read (
            Read_Amt  => Cfloat_Queue_Manager.Nep_Type (
               Cfloat_Queue_Manager.Size (QFiltered_Data)),
            Offset_Amt => 0,
            From_Queue => QFiltered_Data));
      --|
      --|
      --| Consume data from the PID's output queues.
      --|
      Cfloat_Queue_Manager.Consume (
         Amount => Cfloat_Queue_Manager.Nep_Type (
            Cfloat_Queue_Manager.Size (QFiltered_Data)),
         Queue => QFiltered_Data);
   end Pid;
end Io_Board;
```

Ada Test Procedue for IO_BOARD

```
--+--------------------------------------------------------------------------
--| File: io_board_test.ada
--| Generated by the MCCI Graph Translation Tool (GrTT) - Version: 1.0
--| On 04/25/96, at 16:07:56
--+--------------------------------------------------------------------------
--|
--| With Clauses for Basic GrTT Types
with Cfloat_Type_Package;
with Cint_Type_Package;
with Float_Type_Package;
with Int_Type_Package;
--|
--| With Clauses for I/O Routines
with Read_Grtt_Inputq_Cint;
with Read_Grtt_Inputq_Float;
```

```
with Read_Grtt_Inputq_Int;
with Write_Grtt_Outputq_Cfloat;
--|
--| With Clause for GrTT PID
with Io_Board;
--|
with Text_Io;

procedure Io_Board_Test is

   Ntaps : Int_Type_Package.Int_Vector_Access_Type;
   Filt_Wts : Float_Type_Package.Float_Vector_Access_Type;
   Sensor_Data : Cint_Type_Package.Cint_Vector_Access_Type;
   Filtered_Data : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
begin
   --|
   --|
   --| Read PID Input Vectors
   --|
   Text_Io.Put_Line ("Reading input file NTAPS.dat");
   Ntaps := Read_Grtt_Inputq_Int ("NTAPS.dat");
   Text_Io.Put_Line ("Reading input file FILT_WTS.dat");
   Filt_Wts := Read_Grtt_Inputq_Float ("FILT_WTS.dat");
   Text_Io.Put_Line ("Reading input file SENSOR_DATA.dat");
   Sensor_Data := Read_Grtt_Inputq_Cint ("SENSOR_DATA.dat");
   --|
   --|
   --| Initialize PID Output Vectors
   --|
   Filtered_Data := new Cfloat_Type_Package.Cfloat_Vector_Type (0..1024);
   --|
   --|
   --| Initialize Queues for PID
   --|
   Io_Board.Init (
      1024,
      1024,
      1024,
      1024);
   --|
   --|
   --| Call to PID Processing Routine
   --|
   Text_Io.Put_Line ("Processing Data");
   Io_Board.Pid (
      Ntaps,
      Filt_Wts,
      Sensor_Data,
      Filtered_Data);
   --|
   --|
   --| Print PID Output Vectors
   --|
   Text_Io.Put_Line ("Writing Output to FILTERED_DATA.dat");
   Write_Grtt_Outputq_Cfloat.Single ("FILTERED_DATA.dat", Filtered_Data);
end Io_Board_Test;
```

ATTACHMENT 4
Software Allocation Graph and Equivalent Application Graph

Software Allocation Graph

```
%GRAPH (SAR

       INPUTQ    = [1..4]YRANGE : CFLOAT

       OUTPUTQ   = [1..4]OUTI : CFLOAT

       )

%GIP (N_P_RANGE : INT INITIALIZE TO 4)

%GIP (N_F_A : INT INITIALIZE TO 64)

%GIP (N_R : INT INITIALIZE TO 236)

%GIP (N_FFT : INT INITIALIZE TO 256)

%GIP (N_FFT_AZ : INT INITIALIZE TO 128)

%GIP (N_P_AZ : INT INITIALIZE TO 4)

%GIP (RCSMUL : FLOAT ARRAY(256)
                  INITIALIZE TO 256 OF 1.0E0 )
       %%Get actual initial values from rcs_kernc.dat

%GIP (VMUL_AZ : CFLOAT ARRAY(128)
                  INITIALIZE TO {128 OF <1.0E0, 1.0E0) )
       %%Get actual initial values from az_kernc.dat

%VAR (VMUL : CFLOAT ARRAY(256)
                  INITIALIZE TO {256 OF <1.0E0, 1.0E0) )
       %%Get actual initial values from taylor_kernc.dat

%VAR (PAD : INT
                  INITIALIZE to 21 )

%QUEUE ( [1..N_P_RANGE,1..N_P_AZ]Y0 : CFLOAT
                  INITIALIZE TO 1024 OF {<0.0E0, 0.0E0>} )

%SUBGRAPH ( [N=1..N_P_RANGE]RANGE
       GRAPH = RANGEJ
       GIP   = VMUL,
               RCSMUL
       INPUTQ = [N]YRANGE
       OUTPUTQ = [N][1..N_P_AZ]Y0
       )

%SUBGRAPH ( [M = 1 .. N_P_AZ]AZI
       GRAPH = AZIDJ
       INPUTQ = [1..N_P_RANGE, M]Y0
       OUTPUTQ = [M]OUTI
```

```
        )

%% PARTITIONING INFORMATION

%PRAGMA(PARTITION
      NAME = [I=1..4]P_RANGE
      SUBGRAPH = [I]RANGE
      PROCESSOR = I860
      )

%PRAGMA(PARTITION
      NAME = [J=1..4]P_AZI
      SUBGRAPH = [J]AZI
      PROCESSOR = I860
      )

%PRAGMA(ASSIGNMENT
      PROCESSOR = CE3
      NODE = [1]P_AZI, [1]P_RANGE
      )

%PRAGMA(ASSIGNMENT
      PROCESSOR = CE4
      NODE = [2]P_AZI, [2]P_RANGE
      )

%PRAGMA(ASSIGNMENT
      PROCESSOR = CE5
      NODE = [3]P_AZI, [3]P_RANGE
      )

%PRAGMA(ASSIGNMENT
      PROCESSOR = CE6
      NODE = [4]P_AZI, [4]P_RANGE
      )

%ENDGRAPH
```

Equivalent Application Graph

```
%GRAPH (SAR_EAG

        INPUTQ     = [1..4]YRANGE : CFLOAT

        OUTPUTQ    = [1..4]OUTI : CFLOAT

        )

%GIP (N_P_RANGE : INT INITIALIZE TO 4)

%GIP (N_F_A : INT INITIALIZE TO 64)

%GIP (N_R : INT INITIALIZE TO 236)

%GIP (N_FFT : INT INITIALIZE TO 256)

%GIP (N_FFT_AZ : INT INITIALIZE TO 128)
```

```
%GIP (N_P_AZ : INT INITIALIZE TO 4)

%GIP (RCSMUL : FLOAT ARRAY(256)
                    INITIALIZE TO 256 OF 1.0E0 )
       %%Get actual initial values from rcs_kernc.dat

%GIP (VMUL_AZ : CFLOAT ARRAY(128)
                    INITIALIZE TO {128 OF <1.0E0, 1.0E0) )
       %%Get actual initial values from az_kernc.dat

%VAR (VMUL : CFLOAT ARRAY(256)
                    INITIALIZE TO {256 OF <1.0E0, 1.0E0) )
       %%Get actual initial values from taylor_kernc.dat

%VAR (PAD : INT
                    INITIALIZE to 21 )

%QUEUE ( [1..N_P_RANGE,1..N_P_AZ]Y0 : CFLOAT
                    INITIALIZE TO 1024 OF {<0.0E0, 0.0E0>} )


%NODE ( [N=1..N_P_RANGE]P_RANGE
       PRIMITIVE = MP_RANGE
       PRIM_IN   = N_P_AZ,
                   N_R,
                   N_FFT,
                   VMUL,
                   RCSMUL,
                   PAD,
                   [N]YRANGE
                            THRESHOLD = N_R - 7
       PRIM_OUT  = [N, 1 .. N_P_AZ]Y0
       )

%NODE ( [M = 1 .. N_P_AZ]P_AZI
       PRIMITIVE = MP_AZI
       PRIM_IN   = N_FFT_AZ,
                   N_P_AZ,
                   N_F_A,
                   N_FFT_AZ/2,
                   VMUL,
                   [1..N_P_RANGE, M]Y0
                            THRESHOLD = N_F_A*N-FFT_AZ/2
                            CONSUME = N_F_A*N-FFT_AZ/(2*N_P_AZ)
       PRIM_OUT  = [M]OUTI
       )

%PRAGMA(ASSIGNMENT
     PROCESSOR = CE3
     NODE = [1]P_AZI, [1]P_RANGE
     )

%PRAGMA(ASSIGNMENT
     PROCESSOR = CE4
     NODE = [2]P_AZI, [2]P_RANGE
     )
```

```
%PRAGMA(ASSIGNMENT
     PROCESSOR = CE5
     NODE = [3]P_AZI, [3]P_RANGE
     )

%PRAGMA(ASSIGNMENT
     PROCESSOR = CE6
     NODE = [4]P_AZI, [4]P_RANGE
     )

%ENDGRAPH
```

ATTACHMENT 5
Behavior Models for Range and Azimuth

Ada Specification for Range Graph (GrTT Produced)

```
--+--------------------------------------------------------------------------
--| File: p_range_.ada
--| Generated by the MCCI Graph Translation Tool (GrTT) - Version: 1.0
--| On 03/21/96, at 09:03:28
--+--------------------------------------------------------------------------
--|
--| With Clauses for Basic GrTT Types
with Cfloat_Type_Package;
with Float_Type_Package;

package P_Range is
   --|
   --| Procedure: Init
   --| Used to initialize all queues in the GrTT PID.  If the
   --| procedure Pid is called before this procedure, the exception
   --| Uninitialized_Pid shall be raised.
   --|
   procedure Init (
      Vmul : in Natural;
      Rcsmul : in Natural;
      Step : in Natural;
      O_B1 : in Natural;
      O_B2 : in Natural;
      O_B3 : in Natural;
      O_B4 : in Natural);
   --|
   Uninitialized_Pid : exception;
   --|
   --|
   --| Procedure: Pid
   --| This procedure shall perform the same functionality as the
   --| input partition graph P_Range.
   --|
   procedure Pid (
      Vmul : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type;
      Rcsmul : in out Float_Type_Package.Float_Vector_Access_Type;
      Step : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type;
      O_B1 : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type;
      O_B2 : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type;
      O_B3 : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type;
      O_B4 : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type);
   --|
end P_Range;
```

Ada Body for Range Graph (GrTT Produced)

```
--+--------------------------------------------------------------------------
--| File: p_range.ada
--| Generated by the MCCI Graph Translation Tool (GrTT) - Version: 1.0
--| On 03/21/96, at 09:03:28
```

```
--+-------------------------------------------------------------------------
--|
--| With Clauses for Basic GrTT Types
with Cfloat_Type_Package;
with Float_Type_Package;
with Int_Type_Package;
with Dint_Type_Package;
--|
--| Use Clauses for Basic GrTT Types
use Float_Type_Package;
use Int_Type_Package;
use Dint_Type_Package;
--|
--| With Clauses for GrTT Queue Managers
with Cfloat_Queue_Manager;

--| With Clauses for GrTT Algorithms
with D_Fanout;
with D_Fft;
with D_Vfill;
with D_Vmul;

package body P_Range is
   --|
   --| GrTT PID type definitions...
   --|
   type Gv_State_Type is range 0..1;
   type Comp_State_Type is range 0..1;
   --|
   --|
   --| GrTT PID Constant & Variable definitions...
   --|
   --| PID State Constants & Variables:
   --|
   Top_Of_Period : constant Gv_State_Type := Gv_State_Type'First;
   Composite_Init : constant Comp_State_Type := Comp_State_Type'First;
   --|
   Gv_State : Gv_State_Type;
   Comp_State : Comp_State_Type;
   --|
   --| Declare Graph Variables...
   N_R : Int_Type_Package.Int;
   N_P_Az : Int_Type_Package.Int;
   N_Fft : Int_Type_Package.Int;
   Pad : Cfloat_Type_Package.Cfloat;
   P_Entity : Dint_Type_Package.Dint_Vector_Type (0..3);
   P : Dint_Type_Package.Dint_Vector_Access_Type
      := new Dint_Type_Package.Dint_Vector_Type'(P_Entity);
   --|
   --| Declare Formal Queues...
   QFill : Cfloat_Queue_Manager.Queue_Type;
   QWind : Cfloat_Queue_Manager.Queue_Type;
   QFfto : Cfloat_Queue_Manager.Queue_Type;
   QRsco : Cfloat_Queue_Manager.Queue_Type;
   QStep : Cfloat_Queue_Manager.Queue_Type;
   QO_B1 : Cfloat_Queue_Manager.Queue_Type;
   QO_B2 : Cfloat_Queue_Manager.Queue_Type;
   QO_B3 : Cfloat_Queue_Manager.Queue_Type;
```

```
QO_B4 : Cfloat_Queue_Manager.Queue_Type;
--|
--| Initialization Flag Declaration...
Initialized : Boolean := False;
--|
--|
function Determine_Gv_Set return Gv_State_Type is
begin
   return 1;
end Determine_Gv_Set;
--|
--|
procedure Init (
   Vmul : in Natural;
   Rcsmul : in Natural;
   Step : in Natural;
   O_B1 : in Natural;
   O_B2 : in Natural;
   O_B3 : in Natural;
   O_B4 : in Natural) is
begin
   --| Initialize State Variables:
   Gv_State := Top_Of_Period;
   Comp_State := Composite_Init;
   --|
   --| Set Initialize Flag
   Initialized := True;
   --|
   --| Initialize Formal Queue Buffers
   Cfloat_Queue_Manager.Initialize (Step, 0, QStep);
   Cfloat_Queue_Manager.Initialize (O_B1, 0, QO_B1);
   Cfloat_Queue_Manager.Initialize (O_B2, 0, QO_B2);
   Cfloat_Queue_Manager.Initialize (O_B3, 0, QO_B3);
   Cfloat_Queue_Manager.Initialize (O_B4, 0, QO_B4);
   --|
   --| Initialize Persistent Queue Buffers (if any)
end Init;
--|
--|
procedure Pid (
   Vmul : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   Rcsmul : in out Float_Type_Package.Float_Vector_Access_Type;
   Step : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   O_B1 : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   O_B2 : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   O_B3 : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   O_B4 : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type) is

   procedure P_Range_Kernel (
      QStep : in out Cfloat_Queue_Manager.Queue_Type;
      QO_B1 : in out Cfloat_Queue_Manager.Queue_Type;
      QO_B2 : in out Cfloat_Queue_Manager.Queue_Type;
      QO_B3 : in out Cfloat_Queue_Manager.Queue_Type;
      QO_B4 : in out Cfloat_Queue_Manager.Queue_Type) is
      --|
      --| Local Data Vector Declarations
      --|
      Fill : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
```

```
   Wind : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   Ffto : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   Rsco : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
begin
   case Gv_State is
      when Top_Of_Period =>
         Gv_State := Determine_Gv_Set;
         Comp_State := Composite_Init;

      --| Graph Value Set #1
      when 1 =>
         N_R := 470;
         N_P_Az := 4;
         N_Fft := 256;
         Pad.Re :=  0.00000000000000E+00
         ;
         Pad.Im :=  0.00000000000000E+00
         ;
         P (0) := 64;
         P (1) := 64;
         P (2) := 64;
         P (3) := 64;
         case Comp_State is
            --| Initialization Composite State
            when Composite_Init =>
               --| Init Formal Thresholds
               Cfloat_Queue_Manager.Set_Threshold (235, QStep);
               --|
               --| State Variable Maintenance
               Comp_State := Comp_State + 1;

            --| Periodic Composite #1
            when 1 =>
               --| Circular Buffer Inits
               Cfloat_Queue_Manager.Initialize (256, 256, QFill);
               Cfloat_Queue_Manager.Initialize (256, 256, QWind);
               Cfloat_Queue_Manager.Initialize (256, 256, QFfto);
               Cfloat_Queue_Manager.Initialize (256, 256, QRsco);
               --|
               --| Pode Executions
               Fill := new
                  Cfloat_Type_Package.Cfloat_Vector_Type (0..255);
               Step := new Cfloat_Type_Package.Cfloat_Vector_Type'(
                  Cfloat_Queue_Manager.Read (235, 0, QStep));
               D_VFILL.Prim (
                  235, 21, 0, Pad, 235, Step, Fill);
               Cfloat_Queue_Manager.Consume (235, QStep);
               Cfloat_Type_Package.Compound_Type_Pkg.Free (Step);
               Cfloat_Queue_Manager.Produce (Fill.all, QFill);
               Cfloat_Type_Package.Compound_Type_Pkg.Free (Fill);
               --|
               Wind := new
                  Cfloat_Type_Package.Cfloat_Vector_Type (0..255);
               Fill := new Cfloat_Type_Package.Cfloat_Vector_Type'(
                  Cfloat_Queue_Manager.Read (256, 0, QFill));
               D_VMUL.Prim (
                  256, 1, 256, Fill, 256, Vmul, Wind);
               Cfloat_Queue_Manager.Consume (256, QFill);
```

```
                Cfloat_Type_Package.Compound_Type_Pkg.Free (Fill);
                Cfloat_Queue_Manager.Produce (Wind.all, QWind);
                Cfloat_Type_Package.Compound_Type_Pkg.Free (Wind);
                --|
                Ffto := new
                   Cfloat_Type_Package.Cfloat_Vector_Type (0..255);
                Wind := new Cfloat_Type_Package.Cfloat_Vector_Type'(
                   Cfloat_Queue_Manager.Read (256, 0, QWind));
                D_FFT.Prim (
                   256, 256, 0, 1, 0, 256, Wind, Ffto);
                Cfloat_Queue_Manager.Consume (256, QWind);
                Cfloat_Type_Package.Compound_Type_Pkg.Free (Wind);
                Cfloat_Queue_Manager.Produce (Ffto.all, QFfto);
                Cfloat_Type_Package.Compound_Type_Pkg.Free (Ffto);
                --|
                Rsco := new
                   Cfloat_Type_Package.Cfloat_Vector_Type (0..255);
                Ffto := new Cfloat_Type_Package.Cfloat_Vector_Type'(
                   Cfloat_Queue_Manager.Read (256, 0, QFfto));
                D_VMUL.Prim (
                   256, 1, 256, Ffto, 256, Rcsmul, Rsco);
                Cfloat_Queue_Manager.Consume (256, QFfto);
                Cfloat_Type_Package.Compound_Type_Pkg.Free (Ffto);
                Cfloat_Queue_Manager.Produce (Rsco.all, QRsco);
                Cfloat_Type_Package.Compound_Type_Pkg.Free (Rsco);
                --|
                O_B1 := new
                   Cfloat_Type_Package.Cfloat_Vector_Type (0..63);
                O_B2 := new
                   Cfloat_Type_Package.Cfloat_Vector_Type (0..63);
                O_B3 := new
                   Cfloat_Type_Package.Cfloat_Vector_Type (0..63);
                O_B4 := new
                   Cfloat_Type_Package.Cfloat_Vector_Type (0..63);
                Rsco := new Cfloat_Type_Package.Cfloat_Vector_Type'(
                   Cfloat_Queue_Manager.Read (256, 0, QRsco));
                D_FANOUT.Prim (
                   256, P, 1, 1, 256, Rsco, New Cfloat_Type_Package.
                   Cfloat_Family_Array_Type'(O_B1, O_B2, O_B3, O_B4),
                   Int_Type_Package.Compound_Type_Pkg.Unused);
                Cfloat_Queue_Manager.Consume (256, QRsco);
                Cfloat_Type_Package.Compound_Type_Pkg.Free (Rsco);
                Cfloat_Queue_Manager.Produce (O_B1.all, QO_B1);
                Cfloat_Type_Package.Compound_Type_Pkg.Free (O_B1);
                Cfloat_Queue_Manager.Produce (O_B2.all, QO_B2);
                Cfloat_Type_Package.Compound_Type_Pkg.Free (O_B2);
                Cfloat_Queue_Manager.Produce (O_B3.all, QO_B3);
                Cfloat_Type_Package.Compound_Type_Pkg.Free (O_B3);
                Cfloat_Queue_Manager.Produce (O_B4.all, QO_B4);
                Cfloat_Type_Package.Compound_Type_Pkg.Free (O_B4);
                --|
                --| Formal Threshold Updates
                Cfloat_Queue_Manager.Set_Threshold (0, QStep);
                --|
                --| State Variable Maintenance
                Gv_State := Top_Of_Period;
        end case;
end case;
```

```
   end P_Range_Kernel;

begin    --| Beginning of Procedure Pid
   --|
   --| Verify that the PID's Queues are initialized
   --|
   if not Initialized then
      raise Uninitialized_Pid;
   end if;
   --|
   --|
   --| Move input data from arrays to queues (circular buffers).
   Cfloat_Queue_Manager.Produce (Step.all, QStep);
   --|
   --| Call PID Kernel Processing.
   while (
      Cfloat_Queue_Manager.Over_Threshold (QStep)) loop
      --|
      P_Range_Kernel (
         QStep,
         QO_B1,
         QO_B2,
         QO_B3,
         QO_B4);
   end loop;
   --|
   --|
   --| Move the output data from the PID's output queues
   --| onto arrays which output the data to the I/O wrapper.
   --|
   O_B1 := new Cfloat_Type_Package.Cfloat_Vector_Type'(
      Cfloat_Queue_Manager.Read (
         Read_Amt  => Cfloat_Queue_Manager.Nep_Type (
            Cfloat_Queue_Manager.Size (QO_B1)),
         Offset_Amt => 0,
         From_Queue => QO_B1));
   O_B2 := new Cfloat_Type_Package.Cfloat_Vector_Type'(
      Cfloat_Queue_Manager.Read (
         Read_Amt  => Cfloat_Queue_Manager.Nep_Type (
            Cfloat_Queue_Manager.Size (QO_B2)),
         Offset_Amt => 0,
         From_Queue => QO_B2));
   O_B3 := new Cfloat_Type_Package.Cfloat_Vector_Type'(
      Cfloat_Queue_Manager.Read (
         Read_Amt  => Cfloat_Queue_Manager.Nep_Type (
            Cfloat_Queue_Manager.Size (QO_B3)),
         Offset_Amt => 0,
         From_Queue => QO_B3));
   O_B4 := new Cfloat_Type_Package.Cfloat_Vector_Type'(
      Cfloat_Queue_Manager.Read (
         Read_Amt  => Cfloat_Queue_Manager.Nep_Type (
            Cfloat_Queue_Manager.Size (QO_B4)),
         Offset_Amt => 0,
         From_Queue => QO_B4));
   --|
   --|
   --| Consume data from the PID's output queues.
   --|
```

```
      Cfloat_Queue_Manager.Consume (
         Amount => Cfloat_Queue_Manager.Nep_Type (
            Cfloat_Queue_Manager.Size (QO_B1)),
         Queue => QO_B1);
      Cfloat_Queue_Manager.Consume (
         Amount => Cfloat_Queue_Manager.Nep_Type (
            Cfloat_Queue_Manager.Size (QO_B2)),
         Queue => QO_B2);
      Cfloat_Queue_Manager.Consume (
         Amount => Cfloat_Queue_Manager.Nep_Type (
            Cfloat_Queue_Manager.Size (QO_B3)),
         Queue => QO_B3);
      Cfloat_Queue_Manager.Consume (
         Amount => Cfloat_Queue_Manager.Nep_Type (
            Cfloat_Queue_Manager.Size (QO_B4)),
         Queue => QO_B4);
   end Pid;
end P_Range;
```

Ada Specification for Azimuth Graph (GrTT Produced)

```
--+-------------------------------------------------------------------------
--| File: p_azi_.ada
--| Generated by the MCCI Graph Translation Tool (GrTT) - Version: 1.0
--| On 03/21/96, at 09:03:03
--+-------------------------------------------------------------------------
--|
--| With Clauses for Basic GrTT Types
with Cfloat_Type_Package;

package P_Azi is
   --|
   --| Procedure: Init
   --| Used to initialize all queues in the GrTT PID.  If the
   --| procedure Pid is called before this procedure, the exception
   --| Uninitialized_Pid shall be raised.
   --|
   procedure Init (
      Yo : in Natural;
      Vmul_Az : in Natural;
      Yoa_Az : in Natural);
   --|
   Uninitialized_Pid : exception;
   --|
   --|
   --| Procedure: Pid
   --| This procedure shall perform the same functionality as the
   --| input partition graph P_Azi.
   --|
   procedure Pid (
      Yo : in out Cfloat_Type_Package.Cfloat_Family_Array_Access_Type;
      Vmul_Az : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type;
      Yoa_Az : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type);
   --|
end P_Azi;
```

Ada Body for Azimuth Graph (GrTT Produced)

```
--+--------------------------------------------------------------------------
--| File: p_azi.ada
--| Generated by the MCCI Graph Translation Tool (GrTT) - Version: 1.0
--| On 03/21/96, at 09:03:03
--+--------------------------------------------------------------------------
--|
--| With Clauses for Basic GrTT Types
with Cfloat_Type_Package;
with Int_Type_Package;
with Dint_Type_Package;
--|
--| Use Clauses for Basic GrTT Types
use Int_Type_Package;
use Dint_Type_Package;
--|
--| With Clauses for GrTT Queue Managers
with Cfloat_Queue_Manager;

--| With Clauses for GrTT Algorithms
with D_Fanin;
with D_Fft;
with D_Mtrans;
with D_Vmul;

package body P_Azi is
   --|
   --| GrTT PID type definitions...
   --|
   type Gv_State_Type is range 0..1;
   type Comp_State_Type is range 0..1;
   --|
   --|
   --| GrTT PID Constant & Variable definitions...
   --|
   --| PID State Constants & Variables:
   --|
   Top_Of_Period : constant Gv_State_Type := Gv_State_Type'First;
   Composite_Init : constant Comp_State_Type := Comp_State_Type'First;
   --|
   Gv_State : Gv_State_Type;
   Comp_State : Comp_State_Type;
   --|
   --| Declare Graph Variables...
   N_P_Az : Int_Type_Package.Int;
   N_F_A : Int_Type_Package.Int;
   N_Fft_Az : Int_Type_Package.Int;
   N_Fft_Az2 : Int_Type_Package.Int;
   Paz_Entity : Dint_Type_Package.Dint_Vector_Type (0..3);
   Paz : Dint_Type_Package.Dint_Vector_Access_Type
      := new Dint_Type_Package.Dint_Vector_Type'(Paz_Entity);
   --|
   --| Declare Formal Queues...
   QSyfc_Az : Cfloat_Queue_Manager.Queue_Type;
   QYfco : Cfloat_Queue_Manager.Queue_Type;
   QY_Az : Cfloat_Queue_Manager.Queue_Type;
```

```
QVmaul : Cfloat_Queue_Manager.Queue_Type;
QYo : Cfloat_Queue_Manager.Queue_Family_Type (0..3);
QVmul_Az : Cfloat_Queue_Manager.Queue_Type;
QYoa_Az : Cfloat_Queue_Manager.Queue_Type;
--|
--| Initialization Flag Declaration...
Initialized : Boolean := False;
--|
--|
function Determine_Gv_Set return Gv_State_Type is
begin
   return 1;
end Determine_Gv_Set;
--|
--|
procedure Init (
   Yo : in Natural;
   Vmul_Az : in Natural;
   Yoa_Az : in Natural) is
begin
   --| Initialize State Variables:
   Gv_State := Top_Of_Period;
   Comp_State := Composite_Init;
   --|
   --| Set Initialize Flag
   Initialized := True;
   --|
   --| Initialize Formal Queue Buffers
   for I in QYo'range loop
      Cfloat_Queue_Manager.Initialize (Yo, 0, QYo (I));
   end loop;
   Cfloat_Queue_Manager.Initialize (Vmul_Az, 0, QVmul_Az);
   Cfloat_Queue_Manager.Initialize (Yoa_Az, 0, QYoa_Az);
   --|
   --| Initialize Persistent Queue Buffers (if any)
end Init;
--|
--|
procedure Pid (
   Yo : in out Cfloat_Type_Package.Cfloat_Family_Array_Access_Type;
   Vmul_Az : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   Yoa_Az : in out Cfloat_Type_Package.Cfloat_Vector_Access_Type) is

   procedure P_Azi_Kernel (
      QYo : in out Cfloat_Queue_Manager.Queue_Family_Type;
      QVmul_Az : in out Cfloat_Queue_Manager.Queue_Type;
      QYoa_Az : in out Cfloat_Queue_Manager.Queue_Type) is
      --|
      --| Local Data Vector Declarations
      --|
      Syfc_Az : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
      Yfco : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
      Y_Az : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
      Vmaul : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   begin
      case Gv_State is
         when Top_Of_Period =>
            Gv_State := Determine_Gv_Set;
```

```
      Comp_State := Composite_Init;

   --| Graph Value Set #1
   when 1 =>
      N_P_Az := 4;
      N_F_A := 64;
      N_Fft_Az := 128;
      N_Fft_Az2 := 64;
      Paz (0) := 64;
      Paz (1) := 64;
      Paz (2) := 64;
      Paz (3) := 64;
      case Comp_State is
         --| Initialization Composite State
         when Composite_Init =>
            --| Init Formal Thresholds
            Cfloat_Queue_Manager.Set_Threshold (2048, QYo (0));
            Cfloat_Queue_Manager.Set_Threshold (2048, QYo (1));
            Cfloat_Queue_Manager.Set_Threshold (2048, QYo (2));
            Cfloat_Queue_Manager.Set_Threshold (2048, QYo (3));
            Cfloat_Queue_Manager.Set_Threshold (128, QVmul_Az);
            --|
            --| State Variable Maintenance
            Comp_State := Comp_State + 1;

         --| Periodic Composite #1
         when 1 =>
            --| Circular Buffer Inits
            Cfloat_Queue_Manager.Initialize (8192, 8192, QSyfc_Az);
            Cfloat_Queue_Manager.Initialize (8192, 8192, QYfco);
            Cfloat_Queue_Manager.Initialize (8192, 8192, QY_Az);
            Cfloat_Queue_Manager.Initialize (8192, 8192, QVmaul);
            --|
            --| Pode Executions
            Syfc_Az := new
               Cfloat_Type_Package.Cfloat_Vector_Type (0..8191);
            Yo (0) := new Cfloat_Type_Package.Cfloat_Vector_Type'(
               Cfloat_Queue_Manager.Read (2048, 0, QYo (0)));
            Yo (1) := new Cfloat_Type_Package.Cfloat_Vector_Type'(
               Cfloat_Queue_Manager.Read (2048, 0, QYo (1)));
            Yo (2) := new Cfloat_Type_Package.Cfloat_Vector_Type'(
               Cfloat_Queue_Manager.Read (2048, 0, QYo (2)));
            Yo (3) := new Cfloat_Type_Package.Cfloat_Vector_Type'(
               Cfloat_Queue_Manager.Read (2048, 0, QYo (3)));
            D_FANIN.Prim (
               32, Paz, 1, 1, New Cfloat_Type_Package.
               Cfloat_Family_Array_Type'(Yo (0), Yo (1), Yo (2), Yo (
               3)), Syfc_Az, Int_Type_Package.Compound_Type_Pkg.
               Unused);
            Cfloat_Queue_Manager.Consume (1024, QYo (0));
            Cfloat_Type_Package.Compound_Type_Pkg.Free (Yo (0));
            Cfloat_Queue_Manager.Consume (1024, QYo (1));
            Cfloat_Type_Package.Compound_Type_Pkg.Free (Yo (1));
            Cfloat_Queue_Manager.Consume (1024, QYo (2));
            Cfloat_Type_Package.Compound_Type_Pkg.Free (Yo (2));
            Cfloat_Queue_Manager.Consume (1024, QYo (3));
            Cfloat_Type_Package.Compound_Type_Pkg.Free (Yo (3));
            Cfloat_Queue_Manager.Produce (Syfc_Az.all, QSyfc_Az);
```

```
Cfloat_Type_Package.Compound_Type_Pkg.Free (Syfc_Az);
--|
Yfco := new
   Cfloat_Type_Package.Cfloat_Vector_Type (0..8191);
Syfc_Az := new Cfloat_Type_Package.Cfloat_Vector_Type'(
   Cfloat_Queue_Manager.Read (8192, 0, QSyfc_Az));
D_MTRANS.Prim (
   128, 64, 8192, Syfc_Az, Yfco);
Cfloat_Queue_Manager.Consume (8192, QSyfc_Az);
Cfloat_Type_Package.Compound_Type_Pkg.Free (Syfc_Az);
Cfloat_Queue_Manager.Produce (Yfco.all, QYfco);
Cfloat_Type_Package.Compound_Type_Pkg.Free (Yfco);
--|
Y_Az := new
   Cfloat_Type_Package.Cfloat_Vector_Type (0..8191);
Yfco := new Cfloat_Type_Package.Cfloat_Vector_Type'(
   Cfloat_Queue_Manager.Read (8192, 0, QYfco));
D_FFT.Prim (
   128, 128, 0, 1, 0, 8192, Yfco, Y_Az);
Cfloat_Queue_Manager.Consume (8192, QYfco);
Cfloat_Type_Package.Compound_Type_Pkg.Free (Yfco);
Cfloat_Queue_Manager.Produce (Y_Az.all, QY_Az);
Cfloat_Type_Package.Compound_Type_Pkg.Free (Y_Az);
--|
Vmaul := new
   Cfloat_Type_Package.Cfloat_Vector_Type (0..8191);
Y_Az := new Cfloat_Type_Package.Cfloat_Vector_Type'(
   Cfloat_Queue_Manager.Read (8192, 0, QY_Az));
Vmul_Az := new Cfloat_Type_Package.Cfloat_Vector_Type'(
   Cfloat_Queue_Manager.Read (128, 0, QVmul_Az));
D_VMUL.Prim (
   128, 3, 8192, Y_Az, 128, Vmul_Az, Vmaul);
Cfloat_Queue_Manager.Consume (8192, QY_Az);
Cfloat_Type_Package.Compound_Type_Pkg.Free (Y_Az);
Cfloat_Queue_Manager.Consume (128, QVmul_Az);
Cfloat_Type_Package.Compound_Type_Pkg.Free (Vmul_Az);
Cfloat_Queue_Manager.Produce (Vmaul.all, QVmaul);
Cfloat_Type_Package.Compound_Type_Pkg.Free (Vmaul);
--|
Yoa_Az := new
   Cfloat_Type_Package.Cfloat_Vector_Type (0..4095);
Vmaul := new Cfloat_Type_Package.Cfloat_Vector_Type'(
   Cfloat_Queue_Manager.Read (8192, 0, QVmaul));
D_FFT.Prim (
   128, 64, 1, 65, 0, 8192, Vmaul, Yoa_Az);
Cfloat_Queue_Manager.Consume (8192, QVmaul);
Cfloat_Type_Package.Compound_Type_Pkg.Free (Vmaul);
Cfloat_Queue_Manager.Produce (Yoa_Az.all, QYoa_Az);
Cfloat_Type_Package.Compound_Type_Pkg.Free (Yoa_Az);
--|
--| Formal Threshold Updates
Cfloat_Queue_Manager.Set_Threshold (0, QYo (0));
Cfloat_Queue_Manager.Set_Threshold (0, QYo (1));
Cfloat_Queue_Manager.Set_Threshold (0, QYo (2));
Cfloat_Queue_Manager.Set_Threshold (0, QYo (3));
Cfloat_Queue_Manager.Set_Threshold (0, QVmul_Az);
--|
--| State Variable Maintenance
```

```
                    Gv_State := Top_Of_Period;
                end case;
            end case;
        end P_Azi_Kernel;

    begin    --| Beginning of Procedure Pid
        --|
        --| Verify that the PID's Queues are initialized
        --|
        if not Initialized then
            raise Uninitialized_Pid;
        end if;
        --|
        --|
        --| Move input data from arrays to queues (circular buffers).
        for I in Yo'range loop
            Cfloat_Queue_Manager.Produce (Yo (I).all, QYo (I));
        end loop;
        Cfloat_Queue_Manager.Produce (Vmul_Az.all, QVmul_Az);
        --|
        --| Call PID Kernel Processing.
        while (
            Cfloat_Queue_Manager.Over_Threshold (QYo (0)) and
            Cfloat_Queue_Manager.Over_Threshold (QYo (1)) and
            Cfloat_Queue_Manager.Over_Threshold (QYo (2)) and
            Cfloat_Queue_Manager.Over_Threshold (QYo (3)) and
            Cfloat_Queue_Manager.Over_Threshold (QVmul_Az)) loop
            --|
            P_Azi_Kernel (
                QYo,
                QVmul_Az,
                QYoa_Az);
        end loop;
        --|
        --|
        --| Move the output data from the PID's output queues
        --| onto arrays which output the data to the I/O wrapper.
        --|
        Yoa_Az := new Cfloat_Type_Package.Cfloat_Vector_Type'(
            Cfloat_Queue_Manager.Read (
                Read_Amt  => Cfloat_Queue_Manager.Nep_Type (
                    Cfloat_Queue_Manager.Size (QYoa_Az)),
                Offset_Amt => 0,
                From_Queue => QYoa_Az));
        --|
        --|
        --| Consume data from the PID's output queues.
        --|
        Cfloat_Queue_Manager.Consume (
            Amount => Cfloat_Queue_Manager.Nep_Type (
                Cfloat_Queue_Manager.Size (QYoa_Az)),
            Queue => QYoa_Az);
    end Pid;
end P_Azi;
```

ATTACHMENT 6
Test Environments for Range and Azimuth

Ada for Testing Range Graph Translation (GrTT Produced)

```
--+-------------------------------------------------------------------------
--| File: p_range_test.ada
--| Generated by the MCCI Graph Translation Tool (GrTT) - Version: 1.0
--| On 03/21/96, at 09:03:28
--+-------------------------------------------------------------------------
--|
--| With Clauses for Basic GrTT Types
with Cfloat_Type_Package;
with Float_Type_Package;
--|
--| With Clauses for I/O Routines
with Read_Grtt_Inputq_Cfloat;
with Read_Grtt_Inputq_Float;
with Write_Grtt_Outputq_Cfloat;
--|
--| With Clause for GrTT PID
with P_Range;
--|
with Text_Io;

procedure P_Range_Test is

   Vmul : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   Rcsmul : Float_Type_Package.Float_Vector_Access_Type;
   Step : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   O_B1 : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   O_B2 : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   O_B3 : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   O_B4 : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
begin
   --|
   --|
   --| Read PID Input Vectors
   --|
   Text_Io.Put_Line ("Reading input file VMUL.dat");
   Vmul := Read_Grtt_Inputq_Cfloat ("VMUL.dat");
   Text_Io.Put_Line ("Reading input file RCSMUL.dat");
   Rcsmul := Read_Grtt_Inputq_Float ("RCSMUL.dat");
   Text_Io.Put_Line ("Reading input file STEP.dat");
   Step := Read_Grtt_Inputq_Cfloat ("STEP.dat");
   --|
   --|
   --| Initialize PID Output Vectors
   --|
   O_B1 := new Cfloat_Type_Package.Cfloat_Vector_Type (0..1024);
   O_B2 := new Cfloat_Type_Package.Cfloat_Vector_Type (0..1024);
   O_B3 := new Cfloat_Type_Package.Cfloat_Vector_Type (0..1024);
   O_B4 := new Cfloat_Type_Package.Cfloat_Vector_Type (0..1024);
   --|
   --|
   --| Initialize Queues for PID
```

```
   --|
   P_Range.Init (
      1024,
      1024,
      1024,
      1024,
      1024,
      1024,
      1024);
   --|
   --|
   --| Call to PID Processing Routine
   --|
   Text_Io.Put_Line ("Processing Data");
   P_Range.Pid (
      Vmul,
      Rcsmul,
      Step,
      O_B1,
      O_B2,
      O_B3,
      O_B4);
   --|
   --|
   --| Print PID Output Vectors
   --|
   Text_Io.Put_Line ("Writing Output to O_B1.dat");
   Write_Grtt_Outputq_Cfloat.Single ("O_B1.dat", O_B1);
   Text_Io.Put_Line ("Writing Output to O_B2.dat");
   Write_Grtt_Outputq_Cfloat.Single ("O_B2.dat", O_B2);
   Text_Io.Put_Line ("Writing Output to O_B3.dat");
   Write_Grtt_Outputq_Cfloat.Single ("O_B3.dat", O_B3);
   Text_Io.Put_Line ("Writing Output to O_B4.dat");
   Write_Grtt_Outputq_Cfloat.Single ("O_B4.dat", O_B4);
end P_Range_Test;
```

Ada for Testing Azimuth Graph (GrTT Produced)

```
--+-------------------------------------------------------------------------
--| File: p_azi_test.ada
--| Generated by the MCCI Graph Translation Tool (GrTT) - Version: 1.0
--| On 03/21/96, at 09:03:03
--+-------------------------------------------------------------------------
--|
--| With Clauses for Basic GrTT Types
with Cfloat_Type_Package;
--|
--| With Clauses for I/O Routines
with Read_Grtt_Inputq_Cfloat;
with Write_Grtt_Outputq_Cfloat;
--|
--| With Clause for GrTT PID
with P_Azi;
--|
with Text_Io;

procedure P_Azi_Test is
```

```
   Yo : Cfloat_Type_Package.Cfloat_Family_Array_Access_Type;
   Vmul_Az : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
   Yoa_Az : Cfloat_Type_Package.Cfloat_Vector_Access_Type;
begin
   --|
   --|
   --| Read PID Input Vectors
   --|
   Text_Io.Put_Line ("Reading input files YO_x.dat");
   Yo := new Cfloat_Type_Package.Cfloat_Family_Array_Type (0..3);
   Yo (0) := Read_Grtt_Inputq_Cfloat ("YO_0.dat");
   Yo (1) := Read_Grtt_Inputq_Cfloat ("YO_1.dat");
   Yo (2) := Read_Grtt_Inputq_Cfloat ("YO_2.dat");
   Yo (3) := Read_Grtt_Inputq_Cfloat ("YO_3.dat");
   Text_Io.Put_Line ("Reading input file VMUL_AZ.dat");
   Vmul_Az := Read_Grtt_Inputq_Cfloat ("VMUL_AZ.dat");
   --|
   --|
   --| Initialize PID Output Vectors
   --|
   Yoa_Az := new Cfloat_Type_Package.Cfloat_Vector_Type (0..1024);
   --|
   --|
   --| Initialize Queues for PID
   --|
   P_Azi.Init (
      1024,
      1024,
      1024);
   --|
   --|
   --| Call to PID Processing Routine
   --|
   Text_Io.Put_Line ("Processing Data");
   P_Azi.Pid (
      Yo,
      Vmul_Az,
      Yoa_Az);
   --|
   --|
   --| Print PID Output Vectors
   --|
   Text_Io.Put_Line ("Writing Output to YOA_AZ.dat");
   Write_Grtt_Outputq_Cfloat.Single ("YOA_AZ.dat", Yoa_Az);
end P_Azi_Test;
```

GRAPH TRANSLATION TOOL (GRTT) USER'S MANUAL

GRAPH TRANSLATION TOOL (GRTT) USER'S MANUAL

1. Scope

1.1 System Identification

This document describes the use of the Graph Translation Tool (GrTT), a software program for translating a signal processing graph, expressed in the Processing Graph Method (PGM), into Ada source code that implements the signal processing embodied by the graph.

1.2 System Overview

GrTT is a software tool for the translation of a signal processing graph specified using the Processing Graph Method (PGM) to Ada source code that implements the signal processing.

GrTT parses the input graph into a set of data structures, creates instantiations of the graph, determines an execution sequence for the graph, and generates Ada source code that implements the signal processing.

Figure 1 displays the functions performed by GrTT. The input to GrTT is a SPGN file which provides a data flow specification of a signal processing program. SPGN is the notational form of a PGM graph. It may be automatically generated from iconic specification by one of two existing graphical editors. The ability to accommodate controls within the translation is a salient feature of executables generated by GrTT. Graph controls alter the data flow rates through graphs, change the topology of the graph, or change the processing performed by any of the domain primitives included in the executable. Controls are specified to GrTT with Graph Variable (GV) sets. Each GV set specified will cause some variation in the execution behavior of the executable.

At the application level, an equivalent node that has the GrTT executable as its primitive replaces the graph or graph segment translated. Ports of an equivalent node are identical to ports of the graph. Execution behavior of equivalent nodes at its ports are identical to the replaced graph. Identical inputs to a graph and an equivalent node will produce identical outputs under all sets of controls specified by the GV sets. The single node may replace the graph in the overall graphical application. Replacing the graph in a larger application with the equivalent node supports graphical or notational level insertion of the GrTT executable back into the application.

The Ada translation implements a primitive for the equivalent node that performs the processing specified by the input graph and each GV set. The translation includes code to interface the Ada executable. Interfacing code reads data from a file, generates a file containing output data, reads controls

and supporting data tables, and consumes input consume amounts from source queues.

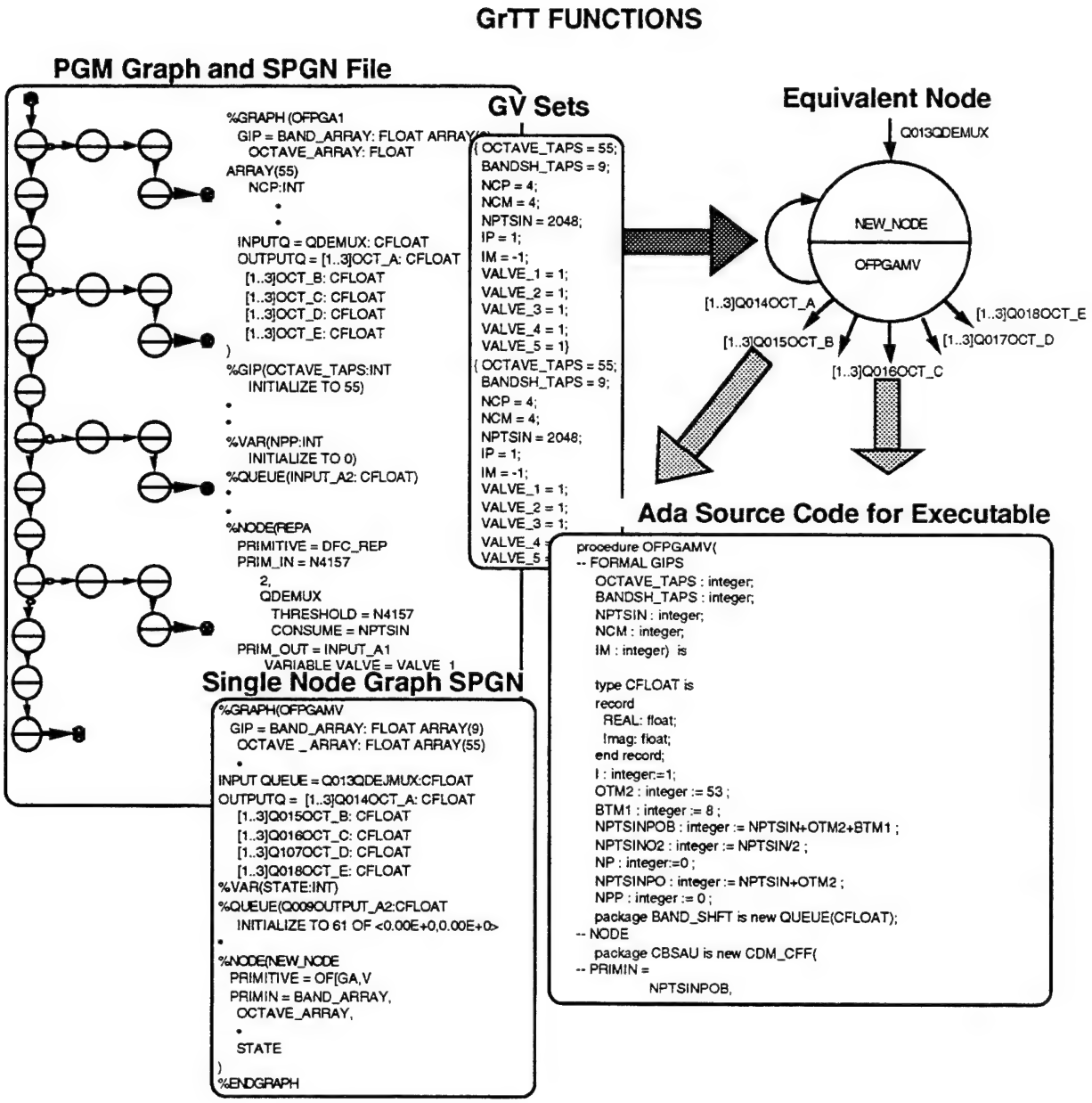


Figure 1 - GrTT Translation Functions

The executable Ada program includes an Ada procedure that implements sequences of domain primitive executions and manages intermediate data. Execution sequences are deterministic for calling the input graph and GV set members. Branching logic is included to select the particular sequence dependent on the GVs passed to the executable in its execution call. The execution sequences may be for a complete graph period (cycle) or a partial

period. If partial periods are implemented, the complete list includes the sequence of parts needed to constitute a complete period and branching logic to select the appropriate part.

GrTT executes on a SUN Workstation under Solaris 2.4. It has a standard UNIX command line interface. Minimal interaction with the user is necessary once GrTT is executing. The Ada source code produced by GrTT must be compiled using the Alsys ObjectAda Compiler version 6.1 in order to correctly link with the library of Ada algorithms provided with GrTT.

Restrictions on graphs that are to be translated into Ada source code are minimal. The major restrictions are:

The input graph must be balanced.

At graph translation time, there must be sufficient information to determine an execution sequence, and the amount of data produced and consumed by each node must be known. Values for formal GIPs and VARs, which are normally provided at instantiation time, will be provided at graph translation time so that these restrictions are met. This means that a graph's execution sequence cannot be dependent on run-time data.

GrTT is based on a defined domain primitive library. This library provides a standardized Application Programmers Interface (API) which is target independent. The primitives in this library represent common signal processing functions with a standardized calling sequence and standardized processing. Because of the exacting nature of automated translation, a special database which contains primitive information translated into a specific format is provided with GrTT. The following primitives are included in the database provided with the initial delivery of GrTT:

D_CAT
D_CDMF
D_CONJ
D_DMUX
D_EMC
D_FANIN
D_FANOUT
D_FFT
D_FIR1S
D_ITOR
D_LIN
D_MAG
D_MMULT
D_MTRANS
D_MUX
D_PWR
D_RTOI
D_SEP

D_STI
D_THRS
D_VADD
D_VDIV
D_VFILL
D_VINP
D_VMUL
D_VSUB

Additional primitives will be added to the database. A list of currently identified and specified domain primitives is provided in Section 3.4.

1.3 Document Overview

Section 3 of this document describes GrTT execution, including input, output and processing options. Section 4 describes how to test GrTT produced source code. The document assumes familiarity with the Processing Graph Method. It is assumed that a user has a working knowledge of PGM programming that is required to generate input graph specifications.

2. Referenced Documents

2.1 Government Documents

SPECIFICATIONS:

Processing Graph Method (PGM) Specification, 15 December 1987, Naval Research Laboratory

STANDARDS:

Defense System Software Development, MIL-STD-2167A, 29 February 1988

Ada Programming Language, ANSI/MIL-STD-1815A, 22 January 1983

Copies of specifications, standards, drawings, and publications required by suppliers in connection with specified procurement functions should be obtained from the contracting agency or as directed by the contracting officer.

2.2. Non Government Documents

The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered a superseding requirement.

"RASSP Primitive Library Standard (Preliminary)" dated 1 February 1995. Management Communications and Control, Inc.

3. GrTT Execution

GrTT is a tool that supports the Processing Graph Method (PGM) by automating the generation of an Ada implementation directly from a Signal Processing Graph Notation (SPGN) form of a signal processing graph. It is necessary to understand both GrTT and its role in PGM to make the most effective use of it. This section describes how a user sets up and executes a GrTT translation.

The current version of GrTT uses the standard Unix command line interface with optional parameters as specified in Section 3.3. GrTT requires two application specific input files and access to the Domain Primitive database. Execution of the Ada source code requires the Alsys ObjectAda Compiler and access to the ObjectAda library of Domain Primitive executables. Both the database and the ObjectAda library are provided with GrTT.

Input files are found using the "environment variable" mechanism supplied by UNIX. The variables expected by GrTT include:

`CM_SPGN_DIR (LOCAL_SPGN_DIR):`
Points to the directory which holds configuration-managed (local) SPGN graphs.

`CM_GVS_DIR (LOCAL_GVS_DIR):`
Points to the directory which holds configuration-managed (local) Graph Variable sets.

`CM_DP_DIR (LOCAL_DP_DIR):`
Points to the directory which holds the configuration-managed (local) domain primitive descriptions and their supporting files.

Note that GrTT's search order gives precedence to local areas over the configuration managed ones. For example, if a user has their own copy of a domain primitive and it is visible on the path specified by the environment variable LOCAL_DP_DIR, the local copy will be used instead of the managed one.

3.1 Input Data

The primary input to GrTT is a graph in the format of PGM. The graph is specified in a file named *graph_name*.GNS. This file contains a description of the graph expressed in Signal Processing Graph Notation (SPGN) which is the language specified by PGM. Other inputs to GrTT are a *graph_name*.gvs file which contains the values for the formal Graph Variables (GV) and Graph Instantiation Parameters (GIP) used by the graph, and the database descriptions of the domain primitives incorporated into the graph. The database descriptions of some of the Domain Primitives are included with GrTT.

3.1.1 SPGN

The graph to be translated is specified using Signal Processing Graph Notation (SPGN) as specified in the PGM Specification.

3.1.2 Graph Value Set

If the input graph has formal GIPs or GVs as inputs that are of integer mode and these are parameters which can possibly impact the execution sequence, a file containing the associated Graph Value Set(s) is required. The Graph Value Set file must contain every set of values for the parameters that will be encountered during the execution of the graph. Each Graph Value Set must contain a value for each required formal GIP and GV, and each Graph Value Set must be enclosed by brackets. Table 3.1 shows an example Graph Value Set file for a graph with two formal GIPs that will accept one of two Graph Value Sets during execution. Graph Value Sets can be used to create "vector NEP variables." When a graph contains nodes with variable NEPs, GrTT cannot under most circumstances determine a static execution sequence and, hence, perform the translation. However, GrTT can perform the translation for a graph with multiple Graph Value Sets by creating multiple states, one for each GV set. This allows the graph writer to have a finite number of Graph Value Sets or "vector NEP variables."

Table 3.1 Graph Value Set

```
{
%GIP ( NODD : INT INITIALIZE TO 2425 )
%GIP ( FG1  : INT INITIALIZE TO 0 )
}
{
%GIP ( NODD : INT INITIALIZE TO 2125 )
%GIP ( FG1  : INT INITIALIZE TO 1 )
}
```

3.1.3 Database Elements

Each domain primitive that will underlie a node in a graph expressed in SPGN that is input to GrTT must have an entry in the Database with information pertaining to the Domain Primitive entered in a GrTT compatible format. The paths to a configuration managed directory and to a working version directory of the Database must be specified prior to GrTT execution. Table 3.2 shows the statements used in a .login file to specify the database being maintained on one computer. (This is done as part of the delivered GrTT script, see the next section.)

Table 3.2 Specifying Database

```
setenv CM_DP_DIR "/home/crobbins/dp"
setenv LOCAL_DP_DIR "./"  # or other directory as needed
```

The CM_DP_DIR refers to the location of a configuration managed directory which should be the path to the database delivered with GrTT and which may contain additions added by the user. The LOCAL_DP_DIR is a working directory to which the user may add new Domain Primitives temporarily subject to the Configuration Management plan of their project. The GrTT program will first search the LOCAL_DP_DIR and then the CM_DP_DIR for the database element required.

3.2 Output

The following output files are produced by GrTT:

The Ada produced source code file which can be compiled in conjunction with the delivered library of Ada algorithms that implement core signal processing routines.

An Ada wrapper that provides for file I/O for reading input data for each input queue and graph variable and writes output data for each output queue and graph variable.

In addition, supplemental outputs, ECOS_GRAPH.LOG and OUTPUT.LOG are produced. These files contain detailed processing information that is used for debugging GrTT. Some of the debug information may be useful for analyzing graphs and the graph translations.

3.3 Command Line Processing Options

GrTT is initiated by a command line which specifies the SPGN and GVS input files as enumerated in sections 3.3.1 and 3.1.2. If the two files have the same name as the graph name with a .GNS file extension for the SPGN input and a .gvs file extension for the GVS input, the command may simply read:

```
grtt -g graph_name
```

GrTT will automatically generate the file names from the graph name and search the appropriate library paths for the files. The library paths for the SPGN files are as follows:

```
setenv CM_SPGN_LIB       "/home/crobbins/graphs"
setenv LOCAL_SPGN_LIB  "./"  #or other directory as needed
```

Alternatively, if the file names differ from the graph name, the .GNS file and the .gvs file can be specified on the command line as follows:

```
grtt -f <any_path/name.GNS>  -v <any_path/name.gvs>
```

Finally, a "usage" mechanism is available which provides an overview of available options from the command line:

```
grtt -h
```

3.4 Domain Primitives

Domain Primitives represent target independent signal processing functions. These functions represent the building blocks for constructing Application Graphs.

A Domain Primitive has three main components: 1) a description that identifies the inputs, output, and processing that are implemented by the primitive, 2) an Ada algorithm that represents the processing implemented by the primitive, and 3) an Autocode Interface that provides a representation that is required for the Autocode tools.

3.4.1 Domain Primitive Description

The Domain Primitive Description is a textual representation of the primitive that provides the information an Application Developer requires in order to incorporate the primitive into an application graph. This information is partitioned into the following sections:

a) Title
b) Functionality
c) Algorithm
d) Input/Output Restrictions
e) Production Function
f) Parameter List
g) See Also

Each of these sections are described below.

3.4.1.1 Title

The Title Section is normally a single line containing an acronym for the primitive and a descriptive title. By convention the acronym starts with "D_." The acronym must be unique within the domain primitive set.

3.4.1.2 Functionality

The Functionality Section consists of a textual description of the processing performed by the primitive. Normally, this is a one or two paragraph overview.

3.4.1.3 Algorithm

The Algorithm Section contains a pseudocode representation of the primitive. The pseudocode should include textual comments. Variables that are not formal parameters should be named with meaningful titles such as "number of executions."

3.4.1.4 Input/Output Restrictions

The Input/Output Restrictions Section describes any restrictions on the input/output parameters. This includes allowable combinations of modes. The default values of any optional parameters are specified in this section.

3.4.1.5 Production Function

The Production Function Section contains expressions for determining the number of output points produced by an execution of the primitive. The expression is normally in terms of the input parameters.

3.4.1.6 Parameter List

The Parameter List Section lists each of the inputs and each of the outputs indicating whether the parameter is required or optional, the use of each parameter (e.g. Number of time samples in input) and the permissible mode(s) of the parameter (int, float, etc.)

3.4.1.7 See Also

The See Also Section is used to refer the User to similar primitives or primitives that are incorporated into this primitive.

3.4.1.8 Example of Domain Primitive Description

The following is an example of a Domain Primitive Description. The Domain Primitive is a one stage fir filter.

D_FIR1S Finite Impulse Response Filter, Single-stage

Functionality

This primitive implements a single-stage finite impulse response filter with NT taps and D:1 decimation. The input consists of a (possibly multiplexed) time series of vectors; if the input is multiplexed the output is also a multiplexed series. If NT > D, the primitive assumes an initialization of MX*(NT-D) elements when the graph begins execution, where MX is the number of time series in the input. It is the user's responsibility to ensure that there is initialization data for each execution of the primitive. The primitive requires an array of NT filter weights.

Algorithm

If the filter weights and the input elements are not of the same precision, the lesser-precision elements will be converted to their greater-precision forms.

```
If NT > D
    number of executions (NE) = (ream(X)/MX - (NT-D))/(N - (NT-D))
    amount of input data processed = (N + (NE-1)*(N-(NT-D)))*MX
    slide between executions = (N - (NT - D))*MX
else
    number of executions (NE) = ream(X)/(N * MX)
    amount of input data processed = NE * N * MX
    slide between executions = N * MX
```

Considering the input to be a contiguous string of data:

```
N  = number of intrinsic elements in one processing vector (for one time
        series), including overlap if present
NT = number of taps
D  = decimation factor
MX      = number of time series in input
A  = weights array
M  = integer((N-NT+D)/D) (output elements per execution)
for ne = 1 .. NE
    for mx = 1 .. MX
        k = mx
        for m = 1 .. M
            Y( (ne-1)*M*MX + (m-1)*MX + mx ) = 0
            for nt = 1 .. NT
                Y( (ne-1)*M*MX + (m-1)*MX + mx ) =
                                Y( (ne-1)*M*MX + (m-1)*MX + mx ) +
                                A(NT+1-nt)*X( (ne-1)*slide*MX + k + (nt-
1)*MX )
            k = k + D*MX
```

If the output is not of the precision in which the calculations are done, the proper conversions are performed.

Input/Output Restrictions

The allowable combinations of input, filter weight and output types are as follows:

X	Y	A
CFLOAT/DCFLOAT	FLOAT/DFLOAT	CFLOAT/DCFLOAT
FLOAT/DFLOAT	FLOAT/DFLOAT	FLOAT/DFLOAT
FLOAT/DFLOAT	CFLOAT/DCFLOAT	CFLOAT/DCFLOAT

The output may be vector or any-dimensional array, as long as there is an integral number of transfer elements in the output.

If MX is unspecified, it is defaulted to 1.
If D is unspecified, it is defaulted to 1.
(N - NT) mod D = 0.

In the case of data overflow, the element will be set to the largest (or smallest) representable value of the type involved.
In the case of data underflow, the element will be set to zero.

Production Function

Vector output

The production function for Y is

```
If NT >= D
    MX*((ream(X)/MX  -  (NT-D))/D)
else
    (ream(X)/N)*(N  -  NT  +  D)/D
```

Array output

The production function for Y is

```
If NT >= D
    (MX*((ream(X)/MX  -  (NT-D))/D))/arraysize(Y)
else
    ((ream(X)/N)*(N  -  NT  +  D)/D)/arraysize(Y)
```

Parameter List

Inputs

N	Number of intrinsic elements in processing vector (including overlap amount) Int
optional MX	Number of time series in input Int
NT	Number of taps Int
optional D	Decimation factor Int

A	Filter weights array (size NT) Floatldfloatlcfloatldcfloat
X	Vector input Floatldfloatlcfloatldcfloat

Outputs

Y	Vector/array output Floatldfloatlcfloatldcfloat

See Also

3.4.2 Domain Primitive Ada Algorithm

The Domain Primitive Ada Algorithm is source code that performs the numerical processing of the primitive. It consists of an Ada Specification and an Ada Body. The Ada algorithm is normally composed of a number of overloaded procedures that correspond to the different input and output modes of the formal parameters.

The Ada Specification is normally implemented as an Ada package named with the acronym of the domain primitive. This acronym is unique amongst the domain primitive set. The package must "with" the packages that contain the data structure formats for the PGM intrinsic modes that can be used as input and/or output modes of the data.

Also included in the specification is the declaration of the procedures which implement the various permissible input and output data modes. By convention all of these procedures are named "Prim." These procedures are overloaded based on actual usage of the primitive.

The Ada Body contains the code for each of the procedures specified in the Ada specification. Each of the procedures implements the processing embodied in the domain primitive including any mode conversions required to format the input and/or output data.

3.4.3 Domain Primitive Autocode Interface

The Autocode Interface component of a Domain Primitive refers to the database element that describes the Domain Primitive in a format which is consistent with GrTT processing. It contains five major sections: 1) the declaration section where the data structures are defined and attributes are specified, 2) the file reference section of a domain primitive which specifies the names of a description file and algorithm file which must be defined for each domain primitive, 3) the produce amount calculation, 4) the timing estimate expression calculation which provides the theoretical Mega-Flops required for any valid

combination of parameter values and data modes, and 5) one or more error check sections for formal parameters.

The syntax for constructing Domain Primitive Autocode Interface is defined in "RASSP Primitive Library Standard (Preliminary)" dated 1 February 1995.

An example of Domain Primitive Autocode Interface for the domain primitive D_FIR1S is contained in that document.

3.4.4 Preliminary Initial Set of Domain Primitives

The list of domain primitives selected as a preliminary initial set is shown in the following table. Detailed User descriptions of each of these primitives can be found in Appendix A to Domain Primitive Library Specification. A copy of this Appendix is available upon request from Management Communications and Control, Inc. (MCCI), 2000 North Fourteenth Street, Suite 220, Arlington, VA 22201.

Vector Mode Conversions

1	D_CTOR	complex	real		Q003, M860	complex vector part separation
2	D_RTOC	real	complex		M860	real vector combination to complex vector
3	D_EMC	int, real, complex	int, real, complex	√	Q003, M860	mode(and precision) conversion
4	D_RTOI	real	int	√		convert real vector to integer vector
5	D_ITOR	int	real	√		convert integer vector to real vector

Vector Binary Primitives

6	D_VADD	int, real, complex	int, real, complex	√	Q003, M860	Vector-vector add
7	D_VDIV	int, real, complex	int, real, complex	√	Q003, M860	vector-vector divide
8	D_VMUL	int, real, complex	int, real, complex	√	Q003, M860	vector-vector multiply and complex conj mult
9	D_VSUB	int, real, complex	int, real, complex	√	Q003, M860	vector-vector subtract
10	D_VINP	int, real, complex	int, real, complex	√	Q003, M860	vector-vector inner product

Vector Comparison Primitives

11	D_CTH2	int, real	int, real		Q003, M860	vector compare and threshold > or < mean vector
12	D_DIFM	int, real, complex	int, real, complex		Q003, M860	vector compare and difference
13	D_THRS	int, real	int, real	√	Q003, M860	vector threshold

Vector Unary Primitives

14	D_CONJ	complex	complex	√	Q003, M860	complex vector conjugate
15	D_PWR	complex	real	√	Q003, M860	complex vector power conversion
16	D_ATAN2	real	real		Q003, M860	arctangent of two vector inputs over [0-2π]
17	D_SINE	real	real		M860	vector sine
18	D_COS	real	real		M860	vector cosine
19	D_TAN	real	real		M860	vector-vector tangents
20	D_INDX	real	real		Q003, M860	vector gather, output selected by control index vec
21	D_LIN	real	real	√	Q003, M860	vector linear scaling [Ax+B]
22	D_LOG	real	real		Q003, M860	vector log [AlogB(X)+C]; B = 2,10
23	D_MAG	real	real	√	Q003, M860	vector magnitude
24	D_RECIP	real	real		M860	vector reciprocal
25	D_VCC2	real	real		Q003, M860	vector upper and lower threshold compare & clip
26	D_SQRT	real, complex	real, complex		Q003, M860	vector square root
27	D_SQR	real, complex	real, complex		M860	vector square
28	D_ZCC	real	real		Q003, M860	vector zero crossing counter
29	D_VFILL	int, real, complex	int, real, complex	√	Q003, M860	vector fill with pad elements
30	D_CPLR	complex	complex		M860	convert rectangular coordinates to polar form
31	D_RECT	complex	complex		M860	convert polar coordinates to rectangular form

Matrix Operations

32	D_MMULT	real, complex	real, complex	√	M860[real]	matrix-matrix multiply
33	D_MTRANS	real, complex	real, complex	√	M860[real]	matrix transpose
34	D_MINV	real, complex	real, complex		M860[real]	matrix invert

Data Conditioning Primitives

35	D_AVG1	real	real		Q003, M860	vector average
36	D_AVGN	real	real		Q003	vector block average
37	D_AVGEXP	real	real		M860	vector exponential average
38	D_DEC	real, complex	real, complex		Q003, M860	vector decimate
39	D_EAVN	real	real		Q003	exponential block averaging filter
40	D_FRQW	complex	complex		Q003	complex frequency weighting
41	D_FRQWC	complex	complex		Q003	Proportional resolution frequency weighting
42	D_HAMN	complex	complex		Q003, M860	Hamming or Hanning weighting
43	D_LINT	real	real		Q003, M860	linear interpolation
44	D_MEF	real	real		Q003	Mean estimation in frequency
45	D_MET	real	real		Q003	Mean estimation in time
46	D_MWAG	real	real		Q003, M860	Sliding window average
47	D_MWGT	complex	complex		Q003	Multiplex weighting
48	D_NME	real	real		Q003	three pass noise mean estimation
49	D_NORM3	real	real		Q003	three pass noise mean estimation
50	D_SMERGE	complex	complex		Q003	concatenate data from multiple queues
51	D_SPL	real, complex	real, complex		Q003	split spectral data into sub bands
52	D_STI	real	real	√	Q003	block averager
53	D_TSS	real, complex	real, complex		Q003, M860	Time series mean, variance,standard deviation
54	D_VDI	real	real		Q003	Variable diagonal averager

Data Format Conversion Primitives

55	D_CAT	real, complex	real, complex	√	Q003	vector concatenate
56	D_DMUX	real, complex	real, complex	√	Q003	vector demultiplex
57	D_DSD	real	real		Q003, M860	data scaling

58	D_FLOC	real, complex	real, complex		Q003	flow control
59	D_LRQT	real	real		Q003	vector requantization with clipping
60	D_MUX	real, complex	real, complex	√	Q003	vector multiplex
61	D_REORD	complex	complex		Q003	reorder and cell selection
62	D_REP	real, complex	real, complex		Q003	replicate
63	D_REQV	real	real		Q003	requantization
64	D_SCAT	int, real, complex	int, real, complex		Q003	selective concatenate
65	D_SEP	int, real, complex	int, real, complex	√	Q003	vector separate
66	D_SWTH	int, real, complex	int, real, complex		Q003	switch
67	D_VCAT	int, real, complex	int, real, complex		Q003	v_array concatenate
68	D_VREP	int, real, complex	int, real, complex		Q003	v_array replicate
69	D_FANOUT	int, real, complex	int, real, complex	√	PIDGen	vector fanout to designated output queues
70	D_FANIN	int, real, complex	int, real, complex	√	PIDGen	vector fanin from designated input queues

FFTs

71	D_FFT	real, complex	real, complex	√	Q003, M860	FFT forward and reverse
72	D_FFT2D	real, complex	real, complex		M860	

Filters

73	D_FIR1S	real, complex	real, complex	√	Q003, M860	1 stage finite impulse response filter
74	D_FIR2S	real, complex	real, complex		Q003	2 stage finite impulse response filter
75	D_IIR1S	real, complex	real, complex		Q003, M860	general infinite impulse response filter

Demodulation and Bandshifting Primitives

76	D_CDMF	real, complex	real, complex	√	Q003	demodulation fixed frequency
77	D_CDMV	real, complex	real, complex		Q003	demodulation variable frequency
78	D_CDMFIR	real, complex	real, complex		Q003	demodulate fixed frequency and FIR filter

Convolutions

79	D_CONVL	real	real		M860	time domain convolution
80	D_CCONVL	real, complex	real, complex		Q003, M860	circular convolution

Image processing

81	D_CONV2D	real	real		M860	3x3 or 5x5 2 D convolution
82	D_SPIN	real	real		M860	image rotate scale and translate

Beamforming Primitives

83	D_BFRF	complex	complex		Q003	Frequency domain beamforming

(1) Vectors may include n dimensional arrays where appropriate
(2) √ indicates Ada routine implementing domain primitive is a GrTT deliverable

4. Testing GrTT Output

All input data files must be created. A separate input data file is required for each input queue and each graph variable. The name of the input file must currently be the name of the formal input queue (in upper case letters) appended with ".dat." There will be an output file created for each formal output queue. The output file will be named "queue_name.dat." The input data files must reside in the current directory when executing GrTT. All output files will be created in the same directory. The output files are compared with data from a previously verified test simulation.

GrTT produces an Ada program which, after compilation, will execute the compiled GrTT produced Ada code that implements the translated graph. This test program reads data from files, executes the code and then writes the output data to files, one for each output entity.

5. Examples

The software delivery tape contains three examples. These are p_azidj, p_rangej, and p_rangej_c1_7. After the software has been properly installed including the setting of environment variables, the user can copy the ".GNS" and the ".gvs" files from the examples subdirectory to a working directory. The example can then be translated by entering the following command line:

```
grtt  -g "graph_name"
```

where "graph_name" is the name of one of the examples.

The translation will produce the following Ada source code files:

graph_name_.ada -- the Ada specification for the translated graph
graph_name.ada -- the Ada body for the translated graph
graph_name_test.ada -- an Ada program to test the translated graph

The examples should translate correctly. By executing the (compiled) test program on the data files contained in the examples subdirectories, the user can test for correct translation. The output data files contained in the directory for the particular example has been validated for the input data included with the example.

If errors occur during the translation of a graph, error messages will be issued to the user. These messages include the type of error. The user can locate the source of the error by examining the following files:

File	Error Type
spgn_yacc.lis	SPGN errors in the input graph, node parameter errors
gvs.lis	Graph Value Set errors
primitive.lis	Primitive parameter errors
name_db_parser.lis	Attempt to call non-existent primitive
ecos_graph.log	Information pertinent to execution of the graph being translated. Useful for identifying NEP errors.